THESE

Présentée à

l'Université de DIJON

pour obtenir le grade

de

DOCTEUR D'UNIVERSITE

Spécialité:

SYSTEMOLOGIE (INFORMATIQUE)

par

MICHAEL J. SPIER

ETUDE SYSTEMOLOGIQUE DE LA MAITRISE DE COMPLEXITE, APPLIQUEE A LA SYNTHESE DES SYSTEMES INFORMATIQUES.

Soutenue le 4 Octobre 1974
Devant la Commission d'Examen

M. FLATO Moshé	Président
M. PIGEAUD Pierre	Examinateur
M. SIMON Jacques	Rapporteur

THESE

Présentée à

l'Université de DIJON

pour obtenir le grade

de

DOCTEUR D'UNIVERSITE

Spécialité:

SYSTEMOLOGIE (INFORMATIQUE)

par

MICHAEL J. SPIER

ETUDE SYSTEMOLOGIQUE DE LA MAITRISE DE COMPLEXITE, APPLIQUEE A LA SYNTHESE DES SYSTEMES INFORMATIQUES.

Soutenue le 4 Octobre 1974
Devant la Commission d'Examen

M. FLATO Moshé	Président
M. PIGEAUD Pierre	Examinateur
M. SIMON Jacques	Rapporteur

UNIV.
LEIDEN
BIBL.

REMERCIEMENTS

Je voudrais remercier et témoigner mon amitié au Professeur Moshé FLATO pour l'intérêt qu'il a porté à ce travail et pour avoir accepté de présider le jury de ma thèse.

Je remercie également le Professeur Pierre PIGEAUD pour avoir accepté de se joindre au jury de ma thèse.

Mes remerciements vont tout particulièrement à Monsieur le Professeur Jacques SIMON pour les discussions très fructueuses que nous avons eues et les remarques pertinentes qu'il m'a faites tout au long de la rédaction de ce travail.

Qu'il trouve ici le témoignage de mon amicale gratitude.

Cet ouvrage est dédié à Françoise Parla qui réalisa l'impossible; encore eut-elle le culot d'insister pour que ce fût du travail bien fait.

Merci.

ETUDE SYSTEMOLOGIQUE DE LA MAITRISE DE COMPLEXITE APPLIQUEE A LA SYNTHESE DES SYSTEMES INFORMATIQUES

MICHAEL J. SPIER

63 Howard Street

Cambridge, Massachusetts 02139

U.S.A.

© Copyright 1974, Michael J. Spier. Tous droits réservés.

TABLE DES MATIERES

PREMIERE PARTIE

ENONCE DU PROBLEME 1

I.1 Introduction Générale 1
I.2 La Nature des Programmes 7
I.3 Concernant la Compréhensibilité des Problèmes . . . 11
I.4 Approche Conceptuelle 14

DEUXIEME PARTIE

THEORIE 19

II.1 Terminologie Informatique 21
II.2 La Fonction Totale 29
II.3 Le Système Général 34
II.4 Le Système Informatique 41
II.5 La Complexité Combinatoire 52

TROISIEME PARTIE

TECHNOLOGIE 64

III.1 Le Principe Relativiste 67
III.2 Concept de la Localité 71
III.3 La Connaissance Locale 75
III.4 Comportement Fonctionnel Prédéterminé 79
III.5 Allocation Dynamique de Mémoire 82
III.6 Liaison Externe Dynamique 85
III.7 Propriétés d'un Système Implémenté 89
III.8 Implémentation Algorithmique 95
III.9 Réduction de la Complexité 102

QUATRIEME PARTIE

CONCEPTION 104

IV.1 Niveaux d'Implémentation 107
IV.2 Décomposition Modulaire 111

APPENDICE 1

LISTE DES OUVRAGES DE L'AUTEUR 116

APPENDICE 2

BIBLIOGRAPHIE RELATIVE A CET OUVRAGE 120

PREMIERE PARTIE: ENONCE DU PROBLEME

I.1: INTRODUCTION GENERALE

Cet ouvrage a pour but d'étudier un problème très réel, urgent et économiquement douloureux dans l'étendue générale de la programmation de calculateurs électroniques. Parmi les milieux industriels et universitaires, les professionnels font face au souci établi et grandissant de la difficulté de contrôler la production, l'entretien et la modification des programmes de gestion d'information*. Du fait de l'importance croissante de l'ordinateur dans la société moderne, le besoin de développer des méthodes technologiques solides, applicables à la production de programmes d'ordinateurs, devient de plus en plus urgent.

La discipline scientifique existante dans le domaine de l'informatique ne semble pas, en fait, nous procurer les éléments intellectuels et techniques suffisantes. Nous proposons ici une approche basée sur une théorie systémologique, nous permettant de formuler un nombre de

---

* Voir références 10 et 44 dans la bibliographie donnée en fin de cet ouvrage.

règles simples et précises, dont l'application systématique conduira à une réduction considérable de la complexité des systèmes informatiques.

Cet ouvrage représente le travail de nombreuses années de recherches et d'expériences --effectuées par l'auteur dans des milieux universitaires autant qu'industriels-- dans la production de software*. Nous avons développé un nombre de techniques et de mécanismes dont certains furent publiés dans la littérature professionnelle. Plus récemment, nous avons mis en application les règles de programmation exposées ici, dans des conditions réelles de production de software industriel et nous avons pu observer et mesurer leurs effets bénéfiques.

C'est un fait bien reconnu que lorsqu'un problème atteint le point où une solution pratique est exigée, il arrive que des chercheurs développent des solutions très voisines, bien souvent en travaillant indépendamment les uns des autres. Les moyens de communication des temps modernes sont si élaborés qu'on ne peut, en toute crédibilité, prétendre ignorer les résultats déjà établis

---

* Nous nous servons du terme anglais "software" pour dénoter les systèmes informatiques intangibles réalisés par programmation d'ordinateurs.

par d'autres. Pourtant, le nombre des publications dont on est inondé est tel qu'il devient pratiquement impossible d'avoir une connaissance totale des activités dans son domaine de recherche. Nous présenterons donc un nombre de principes et techniques de production de software en nous gardant bien cependant de revendiquer quelque droit que ce soit quant à l'origine de leur développement. La bibliographie fournie en fin de cet ouvrage présente un nombre de travaux relatifs au sujet traité dans cette thèse.

Nous estimons que notre contribution majeure tient au fait que ces techniques variées de contrôle de complexité sont présentées sous une perspective théorique uniforme. Nous allons présenter une base théorique systémologique qui, autant que nous sachions, ne fait pas encore partie de la science informatique établie, et suggérons qu'elle pourrait montrer une voie fructueuse pour de futures recherches théoriques.*

Notre attitude est essentiellement pragmatique afin d'établir certains principes technologiques applicables

---

* Voir Wegner (Ref. 53) concernant le manque d'une base scientifique propre à l'informatique, Knuth (Ref. 29) concernant l'aspect algorithmique/mathématique prédominant l'informatique, et Naur (Ref. 42) sur des problèmes nécessitant des solutions non mathématiques.

à la production industrielle de software car leur besoin est réel autant qu'urgent. Du point de vue théorique, notre objectif est de présenter ces principes, d'une manière aussi rigoureuse que possible, de façon à nous permettre d'étudier et d'évaluer leur utilité.

Nous avons pris la décision prudente de poser la fondation de cette théorie systémologique sous la forme d'un certain nombre de définitions axiomatiques. Les systèmes, quelque soit leur nature (informatique, planétaire, biologique, sociologique, moléculaire, culturelle, nucléaire, etc.), possèdent certaines propriétés invariantes qui ont été largement observées et étudiées parmi de nombreuses disciplines de sciences naturelles.* Si nous avions voulu développer, d'une façon compréhensible et rigoureuse, une Théorie Générale des Systèmes, il nous aurait fallu avoir une connaissance de base interdisciplinaire très etendue, que nous ne possédons pas. Nous pensons avoir pris une approche raisonnable, en formulant une base axiomatique particulière à l'informatique, où nous pouvons l'évaluer et la vérifier par expérimentation.

---

* Nous nous basons sur les études systémologiques interdisciplinaires étendues de Allport (Ref. 1), Bertalanffy (Ref. 8, 9) et Laszlo (Ref. 30-32).

La science d'informatique existante (théories de calcul, d'information, d'automate, etc.) se rapporte principalement à des solutions à propriétés algorithmiques. Par opposition, notre recherche concerne les propriétés structurales des systèmes de software. Nous ferons par conséquent une différence très nette entre les deux aspects indépendants de la programmation d'ordinateurs, ceux-ci étant:

1) L'organisation structurale d'un problème logique décomposé en éléments fonctionnels proprement reliés d'une façon contrôlable et compréhensible.

2) La réalisation algorithmique d'un tel élément donné en forme de programme.

Nous aborderons le problème exclusivement sous le premier aspect, celui de la structure systémique, sans nous appuyer sur la nature algorithmique spécifique d'aucun système particulier.

Notre but est d'avoir une connaissance générale des propriétés fondamentales des systèmes de software de façon à pouvoir proposer certaines techniques qui pourraient être appliquées méthodiquement à la conception et à la

réalisation algorithmique de tels systèmes. Vu que la production de hardware* d'ordinateurs (qui à notre avis peut se concevoir comme une "programmation à transistors") a énormement bénéficié du développement de technologies basées sur les concepts de modularité, de standardisation de format et de connectivité uniforme, nous proposons d'examiner avec soin le développement de technologies analogues applicables à la production de software.

---

* Nous nous servons du terme anglais "hardware" pour dénoter les systèmes informatiques composés de dispositifs électroniques tangibles.

I.2: LA NATURE DES PROGRAMMES

En lui-même, l'ordinateur est un appareil inutile. Semblable à une feuille de papier vierge qui doit être empreinte d'information, l'ordinateur doit être programmé de façon à effectuer le calcul désiré. Un programme est la représentation mécanique d'une relation fonctionnelle entre une valeur d'origine donnée et une valeur finale désirée.

La construction d'un appareil de calcul mécanique était --dans un temps pas si lointain-- le vrai projet difficile et monétairement coûteux, et la programmation nécessaire pour rendre cette machine capable d'effectuer un calcul désiré était considérée comme une activité secondaire. Cette situation a changé d'une manière des plus dramatiques et troublantes. Aujourd'hui, la fabrication d'appareils de calcul puissants et sophistiqués est devenue une technologie industrielle bien disciplinée; on peut presque dire que le seul vrai problème auquel le fabricant d'ordinateurs se heurte de nos jours est celui d'offrir un produit plus attrayant et moins cher capable de concurrencer les produits rivaux.

Sans pour cela prétendre que la conception et réalisation d'un ordinateur est une tâche triviale,

on peut admettre que les techniques pour mener à bien un tel projet sont connues. La préoccupation majeure des constructeurs d'ordinateurs est, à l'heure actuelle, de fabriquer des machines plus puissantes à un meilleur prix.

Les programmes réalisés pour les tous premiers ordinateurs étaient presque exclusivement de nature mathématique, où le mathématicien traduisait certaines opérations manuelles --qui auraient pris des jours, des mois, ou même des années-- dans une forme compréhensible à l'ordinateur, lui permettant de réaliser les mêmes calculs à des vitesses électroniques.

Les ordinateurs d'aujourd'hui sont utilisés de moins en moins pour des buts purement mathématiques et de plus en plus pour des fonctions diverses. Ils gouvernent le complexe industriel qui auparavant était contrôlé manuellement; ils calculent la paye; gèrent les livres et comptes en banque; font des réservations d'hôtel, de voiture de location et de billets d'avion; ammassent quotidiennement des milliers d'articles et de petites annonces pour produire automatiquement une composition de journal prête pour la presse, etc. Le nombre

des applications non mathématiques propres aux ordinateurs d'aujourd'hui n'est borné que par l'ingéniosité des techniciens et par des considérations financières de rentabilité.

Toutes ces applications doivent être programmées et, en contraste aigu avec les applications mathématiques, les problèmes de gestion d'information de nos jours ne sont pas très bien définis parce qu'ils n'obéissent pas aux lois d'une discipline scientifique formelle et rigoureuse, mais plutôt aux lois intuitives de la convenance humaine, à sa tradition et satisfaction propre. Par conséquent, un programme réalisé pour la gestion d'information non mathématique n'est plus la représentation d'une fonction scientifique rigoureuse, mais plutôt un très large conglomérat fait d'une multitude de détails divers (d'ailleurs parfois contradictoires).

Il n'est pas difficile de percevoir que plus les applications d'ordinateurs deviennent liées à la manière spontanée et irrationnelle que nous --les humains-- avons choisie pour mener notre vie, plus il devient difficile d'empreindre à l'ordinateur ces modes de comportement que nous considérons comme normaux. Le problème tient au fait que les ordinateurs ne sont que des machines

qui suivent litéralement et rigoureusement les règles qui leur sont données sous forme de programmes. Si la logique d'un programme contient des fonctions contradictoires, un tel dilemme doit être analysé et programmé. Tout cas logiquement possible qui n'aura pas été explicitement prévu dans le programme, pourra, à un moment donné, se présenter au cours de l'activité du programme, provoquant un comportement irrationnel et imprévisible de l'ordinateur.

En tant que membres de l'industrie, à raisonnement pragmatique, nous sommes forcés d'accepter le fait que nos produits doivent satisfaire les besoins du client, et posons la question: est-il possible de développer de nouvelles disciplines de science informatique qui nous aideraient à maîtriser ce problème? Nous estimons que l'approche basée sur la théorie systémologique présentée dans cette étude pourrait nous procurer la base conceptuelle nécessaire pour nous permettre de comprendre ces difficultés et de développer des méthodes suffisantes pour maîtriser cette forme de complexité rencontrée dans les systèmes informatiques.

I.3: CONCERNANT LA COMPREHENSIBILITE DES PROBLEMES

C'est grâce à notre cerveau que nous pouvons reconnaître, définir et résoudre un problème, Contrairement à l'ordinateur digital qui est une machine dont le mode d'opération est de nature binaire (vrai ou faux, 1 ou 0), le cerveau humain possède le remarquable pouvoir d'assimiler des termes relatifs tels que "presque", "peut-être", "similaire" etc., et de percevoir une juste solution par un procédé non rationnel que nous appelons "l'intuition".* Le langage dans lequel cette étude est écrite apporte une information compréhensible au lecteur; jusqu'à présent, nous n'avons pas encore réussi à faire comprendre à l'ordinateur les langages naturels (langages des humains). Il nous faut inventer des formalismes, non naturels aux humains, de façon à communiquer nos idées aux ordinateurs. Bien que le cerveau humain soit remarquablement capable d'assimiler le détail, cette faculté est toutefois limitée. Notre cerveau ne pouvant assimiler qu'un volume de connaissances bien limité, une fois ce seuil dépassé, nous ne pouvons plus comprendre le sens d'une masse d'information. Ceci s'applique à l'art de

---

* Voir Johnson (Ref. 26), Minsky (Ref. 41), Naur (Ref. 42), Spinelli (Ref. 49) et Sussman (Ref 50).

la programmation. Un programmeur ne peut formuler un programme correct que jusqu'à un certain niveau de complexité du problème. Une fois ce seuil dépassé, il n'en est plus capable.

Ce sont vers des fonctions d'applications algorithmiques, comprises dans la capacité intellectuelle de notre cerveau, que se tourne la plupart de la science informatique établie. Nous sommes pourvus de méthodes scientifiques nous aidant à déterminer si un problème donné est soluble ou calculable, si ce calcul sera achevé en un temps fini, et sommes également pourvus de méthodes variées nous permettant d'améliorer la précision des résultats, ou de diminuer le temps d'exécution d'un algorithme donné, etc.*

Nous faisons la distinction entre un problème et sa solution. Nous définissons un problème comme étant compréhensible s'il est intellectuellement assimilable, de façon telle que notre cerveau puisse le décomposer et le réduire en un énoncé rationnel et lucide. Nous définissons un problème comme étant incompréhensible s'il n'est pas intellectuellement assimilable.

---

* Voir Knuth (Ref. 29), et Wegner (Ref. 53, 54).

Ceci n'a absolument rien à voir avec la solution du problème. Un problème peut n'avoir aucune solution, ou sa solution peut varier d'une difficulté extrême à une simplicité ridicule. Toujours est-il, cependant, qu'à moins qu'un problème soit compréhensible, il est inutile de se préoccuper de sa solution. Nous avons identifié trois raisons pour lesquelles les problèmes deviennent incompréhensibles:

Raison 1: MANQUE DE DEFINITION - Si le problème est clairement défini, nous avons une chance de le résoudre. S'il n'est pas défini, nous pouvons au plus nous engager intuitivement dans une série d'essais avec l'espoir de tomber accidentellement sur la solution, catégorie dont fait actuellement partie la plupart de notre programmation.

Raison 2: VOLUME - Quand l'exposé d'un problème comprend une quantité énorme de détails dépassant notre capacité mentale, le problème s'avère être intellectuellement inassimilable.

Raison 3: IRRATIONALITE - Nous utilisons le terme dans son sens mathématique: "incapable d'être exprimé comme nombre entier ou quotient de nombres entiers".

Nous considérons que ce terme est légitime lorsqu' appliqué aux problèmes complexes qui ne peuvent être assimilés par notre cerveau en termes d'unités logiques plus petites, essentiellement indépendantes.

Cet ouvrage traite de la classe de programmes constituants des problèmes irrationnels, indéfinis, à masse de détails immense. Nous les considérons comme des systèmes logiques. En étudiant les propriétés des "systèmes", où le terme "système" s'applique à la programmation d'ordinateurs, nous proposerons certaines techniques par lesquelles la rationalité pourrait être introduite dans les solutions algorithmiques d'une façon systématique. Notre but est d'explorer des méthodes par lesquelles d'immenses programmes pourraient être décomposés en modules logiques plus petits, et si nécessaire, ces derniers en sous-modules encore moindres, etc. jusqu'à ce que chaque module logique ait une taille intellectuellement assimilable, et devienne compréhensible au programmeur, et correctement réalisable.

Nous regrettons de n'avoir aucune manière concise pour établir la dimension ou même pour quantifier le concept "intellectuellement assimilable". Nous étudions un sujet qui est intimement lié à l'intelligence, à la

créativité et à l'intuition humaine, et suggérons que le psychologue pourra y trouver un domaine de recherches intéressantes. De notre point de vue pragramtique, toutefois, nous ne voyons aucune difficulté car les programmes de taille suffisamment réduite (50-100 lignes de programmes) sont aisément maîtrisés par le programmeur moyen. Nous acceptons, par conséquent, l'hypothèse qu'il existe un seuil de simplicité de problèmes en dessous duquel il est possible d'arriver à des solutions algorithmiques acceptablement correctes.

I.4: APPROCHE CONCEPTUELLE

Comme nous le verrons plus loin, le concept de "système" est universel, et les sciences naturelles consistent essentiellement en l'observation, l'identification et la compréhension des systèmes naturels existants. On peut donc dire que la préoccupation primordiale du chercheur en sciences naturelles est essentiellement de nature analytique.

L'ingénieur fait la synthèse des systèmes (machines, ponts, bâtiments); cependant, la complexité de ces systèmes est moins grande que celle des systèmes informatiques. L'ingénieur synthétise des systèmes tangibles à partir de composants physiques naturels dont l'existence même impose certaines contraintes rigoureuses pour l'usage auquel ils peuvent être soumis. Par contre, les systèmes informatiques sont des abstractions logiques existant dans un milieu spatio-temporel artificiel et arbitraire (dans l'univers abstrait de l'ordinateur, il nous est possible de contrôler arbitrairement la nature de l'abstraction interne de "temps"; d'avoir plusieurs déroulements autonomes de temps; de commencer, arrêter, accélérer ou ralentir; même de remonter le temps!).

Le synthétiseur possède donc une liberté surprenante à produire le genre de système de son choix, alors qu'en même temps il n'est soumis qu'à très peu de contraintes l'empêchant de formuler des systèmes à caractère erroné. Etant donné la nature abstraite et intangible de ces systèmes, il devient très difficile de comprendre intellectuellement leur nature fondamentale. Cet ouvrage s'adresse à l'étude des aspects fondamentaux des systèmes informatiques. Nous avons choisi l'approche conceptuelle suivante:

(1) THEORIE - Le développement d'une théorie systémologique formant la perspective subjective par laquelle nous choisissons d'observer et d'étudier ce sujet.

(2) TECHNOLOGIE - La proposition de techniques dont l'application systématique à la synthèse des systèmes informatiques conduit --du point de vue de notre théorie systémologique-- à des systèmes moins complexes.

(3) CONCEPTION - La proposition de certaines méthodes permettant à l'architecte informatique d'utiliser

cette théorie et cette technologie pour produire des systèmes de software d'une qualité meilleure.

Nous voudrions ajouter que nous sommes conscients du fait que la valeur relative de toute perspective théorique ne peut être réellement jugée que par les effets de son application pratique. Il est plus facile d'évaluer l'utilité d'une théorie en sciences physiques, par des expériences, que celle d'une théorie en sciences naturelles non physiques (économiques, sociologiques) où l'expérience conclusive reproduisible se montre impraticable ou même impossible. A cet égard, nous avons eu la chance de pouvoir mettre nos idées en pratique dans le milieu industriel de Digital Equipment Corporation et avons pu observer et mesurer leur effet nettement bénéfique.

DEUXIEME PARTIE: THEORIE

Comme mentionné plus haut, nous nous appuyons sur une fondation théorique systémologique. Le concept de "système" est universel et tout système donné présente certaines propriétés observables, qui sont invariantes quelque soit sa nature intrinsèque (logique, nucléaire, sociologique, moléculaire, mécanique, etc.). Lorsque nous entreprenons d'utiliser une terminologie précise pour définir un système et ses composants, nous sommes confrontés avec de sévères problèmes sémantiques. Il est approprié de dire qu'un système est "une unité structurée composée d'éléments reliés, s'influençant mutuellement à travers leurs liens". Si nous nous représentons une horloge comme étant un système mécanique, il est facile de considérer les ressorts et engrenages intérieurs comme étant les "éléments", l'enchaînement mécanique (c'est à dire les dents de l'engrenage) comme étant le "lien", et la force mécanique comme étant "l'influence".

Cette terminologie est sémantiquement incongrue si nous essayons de l'appliquer à un système moléculaire (nous comprenons les termes "élément" et "influence",

mais qu'est exactement le "lien"?) ou sociologique, ou, pour autant, au système réalisé par la programmation d'ordinateurs. Nous croyons qu'en essayant de surpasser l'étendue du sujet délimité de cette étude, en proposant une terminologie générale pour notre théorie systémologique, nous ne ferions que créer confusion. Nous avons par conséquent choisi de définir d'abord la terminologie informatique nécessaire à notre ouvrage, pour ensuite composer les définitions de notre théorie systémologique en termes de cette terminologie propre aux ordinateurs.

En conséquence, nous ne pouvons vraiment certifier de la validité générale de notre fondation théorique en dehors du domaine de l'informatique. Cependant, comme nous l'avons déjà mentionné, cette fondation est d'une nature généralement acceptable, basée sur des observations scientifiques bien reconnues. Nous voyons donc notre contribution principale comme reposant sur l'identification d'éléments systémiques et sur l'assertion d'une terminologie sémantique acceptable au chercheur scientifique tourné vers l'étude de l'informatique.

II.1: TERMINOLOGIE INFORMATIQUE

Il suffira pour nos besoins de penser à une fonction f comme étant une relation b=f(a) entre les éléments a d'un ensemble A étant le domaine* de la fonction, et les éléments b d'un ensemble B étant le domaine transformé* de la fonction, de telle sorte que pour chaque élément a_i il existe un élément b_i correspondant. La fonction f n'est définie pour aucun élément x n'appartenant pas à l'ensemble A.

L'ordinateur est une machine conçue pour la gestion d'information qui prend une valeur d'entrée, pour la transformer en valeur de sortie. Le domaine autant que le domaine transformé de nos fonctions sont des ensembles de valeurs. Une valeur peut être déposée à l'intérieur de dispositifs de mémoire, que nous appellerons variables, et peut en être extraite. Une variable ne peut contenir qu'une seule valeur à la fois; on peut donc considérer la variable comme la représentation de l'ensemble auquel

---

* Nous nous servons des termes "domaine" et "domaine transformé" pour désigner les termes anglais "domain" et "range".

cette valeur appartient, et utiliserons les notations A et B pour nous référer aux variables représentant des ensembles A et B, respectivement.

Dans la programmation de l'ordinateur, notre objectif est d'obtenir la représentation mécanique de la relation a=f(b) de façon à ce que l'effet de la représentation de f sur la représentation de a résulte en la représentation de b. En terme informatique, la représentation F d'une fonction f est nommée algorithme (ou simplement "programme").

Nous avons raisonné plus haut sur la différence entre la programmation d'une application mathématique et celle d'une application générale de gestion d'information. Dans le cas typique d'une programmation mathématique, une fonction f avec son domaine A est donnée, et l'effort de programmation se limite à la synthèse d'un algorithme F approprié pour calculer les valeurs du domaine transformé B (par exemple pour calculer une table de logarithmes, ou pour calculer π à un degré de précision désiré). Notre seul problème intellectuel est de créer une représentation algorithmique F convenable de la fonction f. Cette classe de problèmes exprimable en termes de fonction et domaine n'est pas le but de cette étude.

La classe de problèmes à laquelle nous sommes primordialement intéressés est normalement exprimée en termes d'entrée (mot familier pour domaine) et de sortie (mot familier pour domaine transformé), où le problème est d'abord de définir la fonction f, et ensuite de programmer d'une façon acceptable une représentation algorithmique F de la fonction. Par exemple, dans le cas typique de l'application de calcul du salaire des employés d'une entreprise, l'entrée est représentée par des enregistrements variés relatifs à l'identité de l'employé, à sa base de salaire, à ses heures de travail effectuées, à ses heures supplémentaires, aux déductions légales et spéciales, etc. La sortie est représentée par les résultats qui doivent être imprimés sur sa feuille de paye, la liste des fonds devant être déposés dans différents comptes bancaires, les déclarations d'impôts qui doivent être soumises au gouvernement, les livres de comptes maintenus par cette entreprise, etc.

Nous définissons notre problème informatique, dans le contexte de cet ouvrage, en énumérant les éléments d'un domaine A et d'un domaine transformé voulu B pour lesquels la relation f doit être déduite et une représentation algorithmique F de f doit être programmée. Si le problème

est intellectuellement assimilable, il y a normalement peu de difficulté à réaliser le programme désiré. Si le problème est intellectuellement inassimilable, on peut en général constater que c'est dû à une, ou plusieurs des raisons énoncées plus haut.

Pour qu'un problème soit bien défini, il faut d'une part que l'entrée soit bien définie, et d'autre part que la fonction ou la sortie soit bien définie. Si ces conditions ne sont pas réalisées, nous dirons que le problème manque de définition.

Raison 1: MANQUE DE DEFINITION - Un tel manque de définition est souvent caractéristique des systèmes informatiques interactifs où l'entrée (tout au moins en partie) est spontanément fournie par un être humain qui s'engage dans un dialogue avec l'ordinateur. Etant donné que le comportement humain est de nature imprévisible, l'entrée de notre programme est potentiellement indéfinie.

Raison 2: VOLUME - Les successions d'étapes de transformation de valeur nécessaires pour définir la représentation algorithmique F de f sont

nombreuses au point qu'il n'est plus humainement possible de comprendre ce que signifie la fonction f en termes algorithmiques. Par exemple, un compilateur de langage de programmation de haut-niveau (COBOL, PL/1) est bien défini en termes d'entrée (syntaxe et sémantiques), et en termes de sortie (séquences d'instruction-machine générées). Pourtant, le processus nécessaire pour passer du langage source au code généré pouvant être exécuté par la machine est extrêmement long, et implique une solution algorithmique volumineuse, au point de la rendre incompréhensible.

Raison 3: IRRATIONALITE- Nos programmes sont codés pour ordinateurs digitaux séquentiels. Ils exécutent, dans une séquence temporelle, une instruction à la fois. Un problème est rationnel (c'est-à-dire décomposable en unités essentiellement indépendantes) si, à un point intermédiaire de la séquence, on peut décomposer la représentation algorithmique de b=f(a) de façon telle que la première partie de cette transformation corresponde à une fonction c=g(a)*, et que la seconde partie de cette transformation corresponde à la

---

* Où c est égal ni à la valeur a, ni à la valeur nulle.

fonction b=h(c), afin de pouvoir réaliser la relation désirée b=f(a) comme étant une fonction composée b=h(g(a)). Le problème est irrationnel si l'on ne peut établir ces points de décomposition. L'irrationalité est symptômatique dans le cas des algorithmes non linéaires (parallélisme, itération).

Nous disons que le problème est acceptablement énoncé si nous avons d'une part une définition du domaine (les valeurs d'entrée) et d'autre part soit une définition de la fonction soit une définition du domaine transformé (les valeurs de sortie) soit une définition mixte consistant partiellement en la fonction et partiellement en des valeurs de sortie, de façon à ce que le programmeur comprenne la nature algorithmique de ce problème.

Observation 1: Nous sommes conscients du fait que notre critère pour juger si un problème est acceptablement énoncé n'est pas rigoureux, et estimons nécessaire d'ajouter quelques mots de clarification. Nous définissons "problème" comme étant quelque chose qui doit être assimilé et compris par le cerveau humain. Ce critère signifie qu'étant donné un domaine A ainsi que, soit la fonction f, soit le domaine

transformé B, soit même la combinaison de la fonction partielle et du domaine transformé partiel, qui permettrait à notre cerveau d'établir une définition du problème de la forme a=f(b), le problème est alors acceptablement énoncé au point de vue de l'assimilation par le cerveau humain. Notre but est d'étudier les difficultés présentées dans la production de software, et comme établi auparavant, les applications de programmation non mathématiques que nous traitons sont, par nature, liées au comportement humain imprévisible. Prétendre par conséquent, que l'énoncé des problèmes devrait être rigoureusement formel, ne nous avancerait pas beaucoup, étant donné que nous ne serons peut-être jamais confrontés à un tel problème idéal. Notre déclaration correspond donc plutôt à ces cas typiques pour lesquels le programmeur dispose de suffisamment d'information pour comprendre le programme qu'il doit effectuer.

Supposons qu'un problème soit compréhensiblement énoncé en terme d'une fonction b=f(a). Nous dirons en outre que la fonction est bien définie si pour chaque élément a_i du domaine A il existe réellement toujours un élément b_i calculable du domaine transformé B.

Observation 2: Nous voulons implémenter* nos fonctions sur des machines à calculer soumises à des contraintes physiques indéniables. Nous pouvons, par exemple, avoir une fonction dont la valeur d'entrée est interprétée comme une demande valide pour une zone de mémoire physique, et dont la valeur de sortie est l'adresse de cette zone de mémoire. Nous pouvons faire une telle demande qui est parfaitement valide, sauf qu'il n'y a plus de mémoire physique disponible, la taille de mémoire physique d'un ordinateur ayant une limite absolue! La fonction est donc bien définie selon notre terminologie si son exécution sera toujours une réussite; elle n'est pas bien définie si l'exécution peut parfois réussir et parfois échouer, selon des circonstances dérivées de conditions indépendantes de la fonction elle même.

Autrement dit, quand nous définissons la fonction, nous devons prendre en considération non seulement la transformation de valeur désirée, mais aussi les conditions (relatives à l'existence physique de l'ordinateur) qui pourraient affecter le comportement fonctionnel de l'implémentation de cette fonction.

---

* Nous nous servons des termes "implémenter" et "implémentation" (qui sont dérivés de l'anglais) pour désigner la réalisation algorithmique par programmation.

II.2: LA FONCTION TOTALE

Quand nous implémentons une fonction en forme algorithmique, son domaine consiste de valeurs rangées en mémoire (en variables). Ces valeurs ne doivent pas nécessairement être des séquences de nombres entiers naturels. Le domaine de la fonction peut parfaitement consister en un ensemble énuméré arbitraire d'entiers (7, 9, 241, 902, 40387). La variable dans laquelle ils sont rangés et qui représente le domaine de la fonction est un dispositif mécanique capable de représenter, par exemple, tous les nombres entiers allant de 0 à 2^n-1. Il n'est donc pas garanti qu'une variable A contienne une valeur qui soit un élément valide a de l'ensemble A (comme défini dans la liste antérieure) et non pas peut-être un élément x (11, 803) qui ne fait pas partie du domaine A. Egalement (comme il a été montré plus haut) il peut être possible pour une fonction a=f(b) d'être appliquée à une valeur d'entrée valide a, bien qu'il soit quand même impossible de calculer un élément valide b de son domaine transformé, du fait de certaines conditions circontancielles.

Nous disons qu'une fonction est une fonction totale* si elle a toujours une valeur de sortie prévisible quelle

---

* Nous nous servons de la traduction litérale du terme anglais "total function" qui est propre à l'informatique.

que soit la valeur d'entrée donnée ou les conditions circonstancielles. La fonction totale a un domaine transformé supplémentaire que nous nommerons sortie anormale, dont les éléments correspondent à tous les cas n'ayant pas d'éléments correspondants (ou calculables) dans la sortie normale de la fonction.

Une fonction totale $(b|c)=f(a)$* a un domaine A, un domaine transformé normal B et un domaine transformé anormal C. Nous adoptons la convention que c_0 est un élément nul de l'ensemble C. Une fonction totale établit toujours une relation entre son élément d'entrée et soit son élément de sortie normal, soit son élément de sortie anormal, mais jamais à des éléments valides (non nuls) des deux à la fois. D'après notre convention, si $c = 0$, un élément b du domaine transformé normal a été choisi. Sinon ($c \neq 0$) a n'est pas un membre du domaine de la fonction -- ou peut-être l'est-il, mais pour des raisons circonstancielles ne correspond pas à un élément calculable du domaine transformé normal-- dans quel cas l'élément c (non nul) a été choisi.

---

* Nous utilisons la notation "|" pour désigner l'opérateur ou exclusif Booléen, la stipulation étant que soit b soit c est toujours "vrai", mais jamais les deux à la fois.

C'est à nous de définir un domaine transformé anormal C de sorte telle que (1) ses éléments correspondront à tout b arbitraire n'étant pas un élément valide du domaine transformé B, (2) le nombre d'éléments c restera suffisamment petit pour constituer un ensemble fini, délimité et aisément énuméré, (3) chaque élément c_i transmettra --par le fait d'avoir été choisi-- une information précise concernant la raison pour laquelle la sortie normale n'a pas été choisie. Donc, la définition de l'ensemble C est obtenue de manière arbitraire et individuelle pour chaque fonction, et dérive de la nature spécifique de cette fonction.

La représentation algorithmique d'une fonction totale $(b|c)=f(a)$ utilise des dispositifs de mémoires d'ordinateurs (que nous appelons "variables") pour représenter les ensembles A, B, C. Soit par contrainte mécanique (la capacité physique étant insuffisante pour contenir la valeur représentant l'élément d'un ensemble), soit pour des raisons de commodité humaine (nous ne voulons peut-être pas que la valeur réside en une seule variable), nous généralisons notre concept de la représentation d'un ensemble de valeurs en permettant au domaine et au domaine transformé d'être chacun représenté par un

ensemble énuméré de variables distinctes, qui peuvent être disjointes. Le domaine $\underline{A}$ est donc mécaniquement représenté par un ensemble de variables (A_1, $\underline{A}_2$, ..., $\underline{A}_e$) et le domaine transformé normal $\underline{B}$ par un ensemble de variables ($\underline{B}_1$, $\underline{B}_2$, ..., $\underline{B}_s$). Un élément ($\underline{a}$ ou $\underline{b}$) est représenté par la valeur collective (ou disjointe, distribuée) de l'ensemble de variables correspondant. Pour simplifier cet ouvrage, nous choisissons que le domaine transformé anormal $\underline{C}$ soit représenté par une seule variable distincte $\underline{C}$ (ceci par commodité, puisque l'ensemble $\underline{C}$ aurait aussi bien pu être représenté par un ensemble de variables).

La représentation algorithmique d'une fonction totale $(\underline{b}|\underline{c})=\underline{f}(\underline{a})$ peut être de la forme:

$$(\underline{B}_1, \underline{B}_2, \ldots, \underline{B}_s \mid \underline{C}) = \underline{F}(\underline{A}_1, \underline{A}_2, \ldots, \underline{A}_e)$$

D'autre part, nous choisissons pour des raisons pragmatiques une représentation algorithmique plus typique correspondant aux langages de programmation existants*:

---

* Nous utilisons des représentations algorithmiques exprimées en termes formels qui sont utilisés couramment dans la litérature professionnelle informatique, derivés des langages de programmation établis.

$\underline{C}$:= $\underline{F}([\underline{A}_1, \underline{A}_2, \ldots, \underline{A}_e],[\underline{B}_1, \underline{B}_2, \ldots, \underline{B}_s])$;
SI $\underline{C}$ = 0 ALORS
DEBUT

    l'opération désirée a réussi et les variables $\underline{B}$ contiennent une valeur de sortie valide.

FIN;
SINON DEBUT

    l'opération désirée a échoué, la valeur de $\underline{C}$ est un code diagnostique précis, indiquant la raison de l'échec.

FIN;

Nous utiliserons le terme <u>système</u> pour désigner la représentation algorithmique d'une fonction. "Système", dans notre terminologie, correspond à n'importe quel niveau d'implémentation, qu'il soit un système d'exploitation, un programme, un sous-programme, une instruction machine hardware, un circuit intégré, etc. Plus loin, nous étudierons des systèmes qui sont composés de modules internes entreliés. Nous avons choisi cette terminologie afin de rapprocher notre ouvrage du concept informatique établi de la "programmation modulaire".

Notons que la "variable" informatique représente un concept qui est différent de celui de la "variable" mathématique! Comme il sera décrit dans la troisième partie, la variable informatique représente une succession temporelle de valeurs transitoires <u>constantes</u>.

II.3: LE SYSTEME GENERAL

Un système est un ensemble (ou unité) structuré qui (normalement) n'est pas réductible à la somme de ses composants et à sa structure. D'où la phrase populaire: "Un système est plus grand que la somme de ses composants". Un système possède des caractéristiques observables, distinctes et évidentes, qui sont uniques à son entité et qui excèdent la somme de ses composants structurés. Prenons par exemple un système composé d'une toile et de peintures à l'huile de couleurs variées. En eux-mêmes, ces composants sont pour nous sans grand intérêt; si, cependant, ils étaient assemblés par Léonard de Vinci en un système de peinture bien connu comme La Joconde, ce système possèderait une caractéristique esthétique unique, sur laquelle on ne pourrait se méprendre, ayant pour nous une signification bien particulière.

On peut observer dans tout système des propriétés qualitatives et des propriétés quantitatives. Définissons les propriétés qualitatives d'un système comme étant celles qui sont intrinsèques à son unité (la propriété de La Joconde, qui est unique à ce système de peinture). L'unité

aussi bien que l'identité du système sont liées à ses propriétés qualitatives de manière unique, exclusive et inséparable. Si l'unité du système est perturbée, ses propriétés qualitatives comme son identité se trouvent transformées. Les propriétés qualitatives du système sont le reflet de sa structure.

Par élimination, nous définissons les propriétés quantitatives d'un système comme étant toutes celles qui restent invariantes après la dissolution totale de la structure systémique. Pour un tableau de peinture, les propriétés quantitatives sont celles propres à la masse de substances restantes après avoir décomposé La Joconde en ses composants de toile, peintures à huile, etc.

Nous pouvons observer le système de l'extérieur et le concevoir comme une unité à propriétés qualitatives distinctes. En règle générale, nous admettons notre impuissance à comprendre un tel système en termes de ses composants énumérés et de sa structure. Cela n'est pas toujours nécessairement vrai cependant, car dans le cas de systèmes très simples, nous sommes en possession des techniques d'intégration nécessaires et suffisantes qui nous permettent de prédire les propriétés qualitatives de

ce système à partir d'une énumeration donnée de ses composants et de sa structure. Malheureusement, quand la taille de la composition du système augmente linéairement (par addition de composants et de structure), très souvent, la complexité du système croît de façon exponentielle, au point de devenir explosive. Considérons le problème bien connu d'un nombre d'objets qui s'attirent mutuellement et qui gravitent sous l'attraction d'un noyau central commun. Quand le nombre de ces objets est égal soit à un, soit à deux, le système est compréhensible et peut être modelé de façon précise en termes de formules mathématiques. Quand nous ajoutons seulement un objet de plus au système, nous avons un système à trois objets gravitants dont la solution mathématique exacte n'a pas encore été établie.*

Nous avons parlé plus haut de la compréhensibilité des problèmes. Définissons maintenant la complexité systémique. Nous disons que le système est simple si nous possédons les moyens d'intégration suffisants pour prédire ses propriétés qualitatives à partir d'une énumeration donnée de ses composants et de sa structure. Par élimination, nous définissons un système comme étant complexe, s'il n'est pas simple. La définition de simplicité systémique est,

---

* Voir Bertalanffy (Ref. 8, 9) sur l'impuissance des mathématiques établies à résoudre des problèmes très complexes.

en large partie, subjective car elle dépend du pouvoir d'assimilation et d'intégration du détail, propre au cerveau humain. La compréhension d'un système simple représente un problème qui est intellectuellement assimilable, tandis que la compréhension d'un système complexe reste un problème qui est intellectuellement inassimilable. Que nous choisissions de considérer un système comme étant simple ou complexe dépend de la marge d'inexactitude que nous sommes prêts à admettre. Plus on accepte une marge d'inexactitude importante, plus on augmente le nombre de moyens d'intégration qui nous sont disponibles, en admettant des méthodes d'intégration de plus en plus approximatives; l'ultime méthode d'intégration approximative est celle que nous appelons l'intuition.

Ce concept de complexité systémique est essentiel à la substance de notre ouvrage, car nous étudions la synthèse de systèmes à grande complexité algorithmique, où la moindre marge d'inexactitude est absolument intolérable! L'ordinateur est un mécanisme qui obéit littéralement à chaque instruction particulière donnée, que celle-ci soit correcte ou non. Il est facile à notre cerveau d'assimiler le problème "programme de calcul de paye" d'une

manière partiellement rationnelle et partiellement intuitive. L'ordinateur, lui, doit l'assimiler en termes d'instructions très précises, rigoureusement énumérées et arrangées en séquence algorithmique rigoureusement correcte, que le programmeur aura dû d'abord synthétiser. Les relations entre telles séquences d'instructions peuvent être parfois d'une subtilité (parallélisme, récursion) dont la nature, en cas extrême, pourrait être du même genre que les relations et les influences existantes parmi les objets gravitants mentionnés plus haut. Si nous considérons maintenant --dans les systèmes informatiques que nous étudions-- que les nombres (de composants et/ou d'influences) ne sont pas exprimés en termes d'unités telles que 1, 2, 3 mais plus typiquement dans l'ordre de grandeur de 10^4 à 10^7 pour le genre de compilateurs et de systèmes d'exploitation construits de nos jours, les implications sont positivement effarantes. Ceci peut expliquer les raisons pour lesquelles nous (l'industrie de programmation de l'ordinateur) n'arrivons pas à comprendre, définir et implémenter correctement nos programmes.

Nous utilisons le terme tas pour désigner un système qui n'a absolument aucune structure (c'est-à-dire où il

n'y a aucune connexion essentielle liant ses composants les uns aux autres, et où les composants ne s'influencent pas mutuellement, de façon essentielle); par définition, un tas n'a que des propriétés quantitatives. Une boutique d'arts peut contenir un tas de toiles vierges et des tas de peintures différentes. En ajoutant des objets à ces tas, nous pouvons les agrandir; en enlevant des objets, les diminuer, mais la nature indéniable d'un tas de toiles vierges est essentiellement inaffectée, que nous ajoutions ou prenions des objets. Ni le tas de toiles, ni les tas de peinture n'ont des propriétés qualitatives qui peuvent être affectées par l'addition ou soustraction d'objets. Ceci est en contraste avec le système structuré de La Joconde, où l'addition d'autres composants en recouvrant la toile d'une couche de lait de chaux, ou la diminution du nombre de ses composants par grattement de sa surface, affecterait considérablement la nature intrinsèque de ce système.

Ce concept de "tas" nous est très important car si nous pouvions démontrer que l'application de certaines techniques de fabrication de software à la synthèse d'un système informatique garantisse la réduction radicale du nombre des connexions et influences structurales internes à notre système, nous pourrions rendre ce système de plus

en plus semblable à un tas, et par conséquent nous réduirions sa complexité, tout en le rendant de plus en plus insensible aux perturbations. Il est clair que la compréhension et la gestion d'objets indépendants, qui n'influencent pas la nature intrinsèque des uns les autres, est plus simple que la compréhension et la gestion d'objets qui s'influencent mutuellement. Notre but est d'introduire autant d'indépendance modulaire que possible dans les systèmes informatiques compliqués que nous avons à synthétiser.

II.4: LE SYSTEME INFORMATIQUE

Nous présentons notre fondation théorique systémologique en un certain nombre de définitions axiomatiques exprimées à l'aide de notre terminologie propre à la programmation des ordinateurs électroniques.

Définition 1: Un système est une unité composée consistant en modules, variables et valeurs. Les modules peuvent être liés les uns aux autres par des variables communes servant de liens. Les modules sont des entités fonctionnelles potentiellement actives. Leur comportement actif est influencé par les valeurs. Les variables sont des liens (ou connexions) inertes par lesquels les valeurs peuvent être transmises. Une valeur ne peut être transmise qu'à l'aide d'une variable.

Définition 2: Une valeur est une influence abstraite. Les modules et les variables sont des objets. Un système ne peut contenir d'autres objets que des modules et des variables. Un module n'est pas une variable. Une variable n'est pas un module.*

---

* Voir observation 3, en fin de cette section, concernant la nature de la variable.

Définition 3: Un module est toujours un système.

Définition 4: Un système qui est complètement isolé de l'univers qui l'entoure, de telle sorte qu'il ne puisse ni être influencé par des valeurs provenant de l'univers qui l'entoure, ni transmettre à cet univers des valeurs qu'il a lui-même (le système) générées, est appelé un système clos. Par élimination, nous définissons un système ouvert comme étant un système qui n'est pas clos.

Définition 5: Un système contenant des modules qui s'influencent les uns les autres à l'aide de valeurs transmises par des variables communes est appelé un système structuré. Un module qui fait partie de la structure d'un système est un module structural. En vertu de la définition 4, un module structural est un système ouvert. Un système qui ne contient absolument aucune structure est appelé un tas. Tous les modules faisant partie d'un tas se comportent comme des systèmes clos, les uns par rapport aux autres.

Définition 6: Un système a des propriétés qualitatives et quantitatives. Les propriétés qualitatives d'un

système sont le reflet de ses composants structurés, mutuellement influençables. Toute perturbation de l'unité d'un système structuré (perturbation de modules, liens, ou influences inter-modulaires) se manifeste par une redéfinition des propriétés qualitatives du système. Par élimination, on définit les propriétés quantitatives d'un système comme étant toutes les autres propriétés du système, c'est-à-dire celles qui demeurent invariantes après une rupture totale de la structure de système. D'après la définition 5, un tas n'a que des propriétés quantitatives.

Définition 7: Un système ouvert est la représentation algorithmique d'une fonction de transformation de valeur $\underline{b}=\underline{f}(\underline{a})$ qui, en réponse à l'influence d'une valeur d'entrée $\underline{a}$ génère toujours une valeur de sortie $\underline{b}$ qui est une influence.

Définition 8: Les systèmes existent dans un univers spécialisé connu sous le nom d'ordinateur digital séquentiel, qui est la représentation mécanique d'une abstraction de l'univers spatio-temporel. Le composant espace d'un ordinateur consiste en une collection finie de dispositifs de mémoire qui

sont les variables. Le composant temps d'un ordinateur consiste en une collection de un ou plusieurs éléments de calcul (dits "processeurs"), chacun d'eux représentant un seul déroulement unilatéral du temps. Le déroulement du temps d'un ordinateur peut être arbitrairement arrêté et remis en marche. Un système peut être arbitrairement soumis à l'influence du temps, et arbitrairement soustrait de l'influence du temps. Les propriétés quantitatives d'un système sont le reflet de l'existence du système dans l'univers spatio-temporel.

Définition 9: On dit qu'un système ouvert est une représentation active d'une fonction de transformation de valeur d'entrée en valeur de sortie si, et seulement si, il est sujet à l'influence du temps. Un système ouvert qui n'est pas sous l'influence du temps est dit inactif. Un système inactif est un objet inerte qui ne peut ni être influencé, ni générer une influence. D'après la définition 4, un système ouvert inactif se comporte comme un système clos*.

---

* De notre point de vue relativiste, nous nous intéressons seulement aux propriétés observables du système. Nous considérons tout système ayant le comportement d'un système clos, comme étant un système clos subjectif.

Définition 10: Un système est un objet discret ayant un nom unique. Le nom d'un système est une valeur. En invoquant (ou prononçant, appelant) le nom d'un système ouvert, ce système ouvert est rendu sujet à l'influence du temps (est activé). Une valeur d'entrée d'un système ouvert devient une influence effective à l'instant de l'activation, et demeure une influence pendant toute la durée de l'activation. Un système actif peut invoquer le nom d'un système ouvert (y compris le sien). L'invocation du nom d'un système ouvert est toujours relative au déroulement du temps sous l'influence duquel l'invocation a lieu, et invoquer le nom d'un système ouvert est le seul moyen pour un système actif de se soustraire à l'influence de ce déroulement de temps particulier, c'est-à-dire de se désactiver (ceci est un point subtil qui sera expliqué à la fin de cette section). Un système actif peut générer des valeurs de sortie pendant toute la durée de son activité.

Définition 11: Une variable est un objet discret ayant un nom unique. Le nom d'une variable est une valeur. Par référence au nom d'une variable, un système actif peut soit observer

(c'est-à-dire lire) soit redéfinir (c'est-à-dire écrire) la valeur de la variable. L'observation d'une valeur est un acte non destructif, qui ne modifie pas la valeur courante de la variable. La redéfinition d'une valeur est un acte destructif et irréversible dont l'effet est de remplacer la valeur courante par une nouvelle valeur générée par le système actif auteur de la référence. Seul un système actif peut faire référence au nom d'une variable (soit pour observation, soit pour redéfinition); par conséquent, et d'après la définition 10, une variable peut être soumise à l'influence du temps seulement par l'intermédiaire d'un système actif.

Définition 12: Un système ouvert n'a qu'une seule propriété qualitative qui est la manifestation, indépendante de l'espace et du temps, de la fonction a=f(b) qu'il représente (qui, d'après une définition antérieure, peut être une fonction mal définie, bien définie ou totale). La propriété quantitative spatiale d'un système est sa taille, exprimée en quantité de mémoire dont il a besoin. La propriété quantitative temporelle d'un système est la durée correspondante à sa période d'activité.

Définition 13: Une variable n'a que des propriétés quantitatives. La propriété quantitative spatiale d'une variable est la quantité d'information (ou la valeur) qu'elle peut prendre. La propriété quantitative temporelle d'une variable est la durée nécessaire au système actif pour observer ou redéfinir la valeur de la variable.

Récapitulons brièvement sur le sens de ces définitions qui constituent une description relativiste du système informatique tel que nous le voyons du point de vue systémologique.

Nous utilisons le terme "système" pour désigner un programme d'ordinateur ayant des entrées et des sorties. Ce programme est un système composé de modules qui sont appelés --en jargon informatique-- "lignes de programme", "instructions", "sous-programmes", etc. Ces modules sont reliés par des variables communes; ces variables sont les liens par l'intermédiaire desquels les modules reliés échangent des valeurs qui influencent le comportement de ces modules. D'après la définition 3, un module est toujours un système; et en fait, dans la vie réelle, tout module informatique est toujours décomposable en sous-

modules moindres qui eux-mêmes sont reliés par des variables internes à ce module; cette décomposition est indéfiniment récursive.

Un système est la représentation algorithmique d'une fonction, générant des valeurs de sortie en réagissant à l'influence de ses valeurs d'entrée. Les valeurs d'entrée sont réellement des influences, car elles affectent de manière observable le comportement fonctionnel du système. Ceci est particulièrement évident en cas d'implémentations de fonctions non totales, où une valeur d'entrée erronée (qui n'est pas un élément valide du domaine de la fonction implémentée) influence le comportement fonctionnel du système de façon visiblement nuisible (par exemple, sortie erronée, boucle d'itération infinie, "crash"*, etc.).

"L'univers" interne de l'ordinateur a un ou plusieurs déroulements de temps séquentiels. Un déroulement de temps

---

* Le terme "crash" est du jargon informatique reconnu, signifiant une tombée en panne de l'ordinateur qui est normalement due à une erreur software fatale.

(en terme informatique: "processeur"*) peut activer un système. A partir de l'instant de son activation, le système peut lire et écrire des valeurs en faisant référence à des noms de variables, jusqu'au point temporel où il invoque le nom d'un système ouvert, moment auquel il se désactive par le fait d'avoir délégué le déroulement de temps au système invoqué.

Tout système actif est influencé par les valeurs d'entrée qu'il obtient des variables A servant ce système actif de connexions d'entrée. Les valeurs de sortie générées par le système actif sont rangées dans les variables B servant ce système actif de connexions de sortie. Plus tard, ces variables B pourraient servir de connexions d'entrée à un autre système (ou peut-être au même système) qui, à son tour, génèrerait des valeurs de sortie qu'il déposerait dans des variables X qui pourraient servir de connexions d'entrée à un système qui serait activé encore plus tard, etc. Nous avons là un processus d'effets enchaînés, où les résultats de futures activités sont déterminés par l'activité présente.

---

* Le processeur est la partie de l'ordinateur qui effectue l'activation séquentielle --dans le temps-- des instructions d'ordinateur individuelles. Un processeur hardware peut être manipulé afin qu'il se comporte comme une multitude de processeurs autonomes dits "processeurs virtuels". Un processeur virtuel est une abstraction temporelle qui n'est plus identique au processeur hardware duquel elle est dérivée.

Comme déjà mentionné dans la définition 10, l'activité simultanée de plusieurs déroulements de temps autonomes pose des problèmes délicats. Un système est activé par le fait d'être soumis à l'influence du temps; en invoquant le nom d'un système à activer, ce système actif se soustrait seulement au déroulement de temps spécifique sous l'influence duquel l'invocation a été effectuée. Selon la définition 10, le système resterait actif dans le cas où il serait encore sous l'influence simultanée d'un autre déroulement de temps. Un système qui est sous l'influence de plusieurs déroulements de temps sera désactivé (d'après la définition 10), seulement après s'être soustrait à l'influence de chacun de ces déroulements de temps. De toute façon, un tel cas ne devrait jamais arriver, car l'influence simultanée de plusieurs déroulements de temps sur un seul système actif rend imprévisible le comportement fonctionnel de ce système, par raison de l'interférence mutuelle --parmi ces activités simultanées -- dans des variables communes dont les valeurs ne seraient plus prévisibles.

Observation 3: Nous avons choisi, dans la définition 2, de distinguer entre les objets dits "modules" et les objets dits "variables". Cette distinction est

arbitraire; une variable peut être considérée comme un module qui a une valeur de sortie égale à sa valeur d'entrée, n'effectuant aucune transformation de valeur, et étant la représentation d'une fonction de la forme $\underline{a}=\underline{f}(\underline{a})$. En fait, si nous étudions les détails de construction d'une mémoire hardware, nous nous apercevons que cette mémoire est une entité active de nature semblable à celle du processeur hardware. Nous aurions pu définir le module comme étant le seul objet composant du système, et arriver au concept de la variable en la définissant comme étant un module d'un comportement fonctionnel de la forme $\underline{a}=\underline{f}(\underline{a})$. Ceci ne nous aurait pas beaucoup avancé, vu qu'il nous aurait fallu définir la manière par laquelle on pourrait lier des modules les uns aux autres. En considérant "système" d'un point de vue relativiste, nous laissons à l'observateur la décision de choisir ce qu'il considère comme système observé et proposons la règle d'observation où tout composant du système choisi ayant une propriété qualitative observée de la forme $\underline{a}=\underline{f}(\underline{a})$ est, du point de vue de l'observateur, une variable.

II.5: LA COMPLEXITE COMBINATOIRE

Un système complexe est irréductible à la somme de ses composants et de sa structure; c'est-à-dire qu'étant donné l'énumération des composants et de la structure d'un système complexe, il nous est impossible de prédire ses propriétés qualitatives. Traduit en terminologie informatique, étant donné une énumération de modules, de variables et de valeurs d'entrée, sachant de quelle manière les modules sont arrangés en séquence algorithmique et de quelle manière ces modules sont reliés par les variables données, il nous est impossible de prédire de façon précise la propriété qualitative du programme qui est --d'après la définition 12-- le comportement fonctionnel de ce programme.

Mais ceci est justement la nature de l'activité intellectuelle que nous appelons "programmation de calculateurs électroniques" où, étant donné une propriété qualitative désirée $\underline{b}=\underline{f}(\underline{a})$, le programmeur doit synthétiser un système informatique ayant ce comportement fonctionnel prédéterminé. Le programmeur effectue la synthèse en choisissant des modules et des variables, en arrangeant les modules en une séquence algorithmique

et en les reliant par des variables communes. En admettant notre impuissance à comprendre un système complexe existant à partir de l'énumeration de ses composants et de sa structure, combien plus difficile doit-il être de préciser les composants et la structure d'un système complexe inexistant dont on ne connait que les propriétés qualitatives spécifiées!

Plus haut, nous avons énoncé que: premièrement, l'incompréhensibilité d'un système complexe est exponentiellement proportionnelle au volume de ses détails composants; deuxièmement, les systèmes complexes défient le pouvoir intellectuel d'assimilation de détail du cerveau humain; et troisièmement, nous allons proposer des techniques de contrôle de complexité qui auront un effet simplifiant démontrable. Pour cela, nous allons développer une formule mathématique exprimant le nombre maximal de systèmes de structures différentes que l'on puisse synthétiser à partir d'une énumération donnée de modules et de variables. En vertu de la définition 6, les propriétés qualitatives du système -- qui sont le résultat prédéterminé devant être atteint par le programmeur effectuant la synthèse du système-- sont intimement liées à la structure du système, et toute

perturbation de la structure implique une redéfinition de ces propriétés qualitatives. Etudions donc le nombre maximal de structures possibles. S'il est égal à un, le programmeur ne pourra jamais faire d'erreur de synthèse; par opposition, si c'est un nombre extrêmement grand, la probabilité de pouvoir synthétiser la structure voulue devient faible. En nous servant de cette formule, nous pourrons évaluer les effets simplifiants de la technologie que nous proposons.

Supposons que nous ayons m modules de M propriétés qualitatives distinctes, et que nous ayons l variables pouvant servir de connexions intermodulaires, où (selon la définition 13) il n'y a aucune distinction qualitative parmi ces l variables et, par conséquent, n'importe quelle variable peut servir au lieu de toute autre. Supposons également qu'un module puisse-t-être lié à d'autres modules par un nombre de connexions parallèles variant de 0 (qui est une connexion nulle) à p. Comme démontré par Figure II.5.1, il nous suffit de nous servir d'une seule variable afin de pouvoir relier les m modules en une seule structure. Dans nos illustrations, nous nous servons de rectangles pour représenter des modules, et de lignes pour représenter des connexions;

chaque rectangle représente un module distinct, chaque ligne représente une variable distincte.

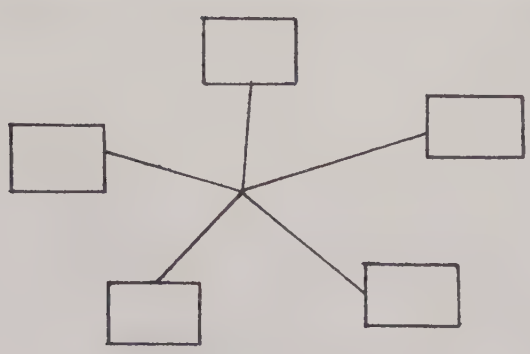

Figure II.5.1

m modules reliés par une seule variable (par contrainte graphique, nous représentons la seule variable par une "ligne" à directions multiples).

Supposons maintenant que tout module est lié à tout autre, et qu'il n'y a qu'une seule connexion entre toute paire de modules; comme démontré par la Figure II.5.2, le nombre de connexions sera $m(m-1)/2$.

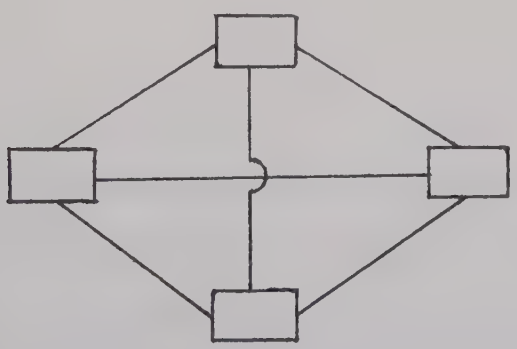

Figure II.5.2

m modules reliés par $m(m-1)/2$ connexions distinctes.

Supposons maintenant que tout module est lié à tout autre, et que chacun d'eux peut avoir p connexions distinctes parallèles. Comme le montre la Figure II.5.3, le nombre de connexions sera pm(m-1)/2.

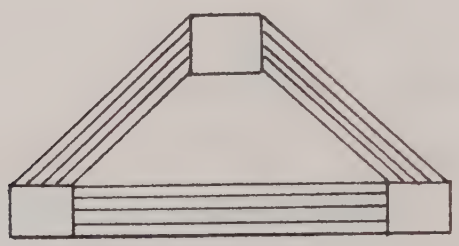

Figure II.5.3

m modules reliés par pm(m-1)/2 connexions distinctes, où m =3 et p = 5.

Pour qu'un système soit structuré, il faut que ses m modules structuraux soient reliés par un minimum d'une variable. Le nombre de connexions distinctes dans un même système structuré peut varier de 1 à pm(m-1)/2.

Examinons maintenant la tâche du programmeur. Il doit choisir m modules, chacun d'entre eux ayant une propriété qualitative (parmi les M propriétés qualitatives données), et les arranger en une séquence algorithmique rigoureuse qui --selon lui-- représentera les

étapes de transformation de valeurs correspondant à la propriété qualitative spécifiée $\underline{b}=\underline{f}(\underline{a})$. Il peut donc établir $\underline{M}^{\underline{m}}$ séquences algorithmiques différentes.

Pour illustrer ceci, imaginons un ordinateur le plus rudimentaire possible, n'ayant que deux modules, chacun d'une propriété qualitative distincte ($\underline{M}=2$). L'un des modules est de la forme LIRE($\underline{V}_i$), l'autre de la forme ECRIRE($\underline{V}_i$). Notre ordinateur possède un registre interne implicite; l'opération LIRE a pour effet d'extraire la valeur se trouvant dans la variable $\underline{V}_i$ et de la déposer dans ce registre implicite. L'opération ECRIRE a pour effet de lire la valeur se trouvant dans le registre implicite et de la déposer dans la variable $\underline{V}_i$ spécifiée.

Programmer cet ordinateur consiste à faire les choix suivants:

1) le choix de $\underline{m}$ modules, chaque module ayant une propriété qualitative distincte, parmi les $\underline{M}$ propriétés qualitatives données.

2) le choix de la séquence dans laquelle les $\underline{m}$ modules sont ordonnés.

3) le choix des variables auxquelles chaque module fait référence.

Imaginons en plus que par sa nature spécifique, notre ordinateur primitif ne tolère pas une connexion nulle de la forme LIRE( ) ou ECRIRE( ), une telle connexion nulle étant une construction syntaxique erronée. Finalement, imaginons que notre ordinateur primitif n'a que deux variables (1=2) entre lesquelles le programmeur peut choisir, et dont les noms sont ALPHA et BETA

Suivons maintenant un petit exercice en essayant de faire le travail du programmeur. Le système à programmer doit avoir une propriété qualitative de la forme a=f(a); spécifiquement, nous devons implémenter le système:

COPIER([BETA],[ALPHA])

dont le comportement fonctionnel doit être de ranger, dans la variable de sortie ALPHA, la valeur qu'il trouve dans la variable d'entrée BETA. La seule façon correcte pour implémenter ce programme sur notre ordinateur est la séquence algorithmique:

LIRE(BETA)
ECRIRE(ALPHA)

Notre formule combinatoire M^m représente toutes les séquences algorithmiques possibles que le programmeur peut choisir. Enumérons ces possibilites:

(1) LIRE / ECRIRE

(2) ECRIRE / LIRE

(3) LIRE / LIRE

(4) ECRIRE / ECRIRE

Figure II.5.4

Enumération de M^m séquences algorithmiques possibles.

Pour obtenir la formule combinatoire exprimant le nombre de structures que nous pouvons synthétiser en tenant compte des choix de variables reliant les m modules, il nous faut considérer le système d'un point de vue relativiste, en distinguant sa structure interne de ses connexions externes (le liant à l'univers qui l'entoure):

1) Connexions Internes: Le programmeur peut choisir parmi l variables internes (dites "locales"), avec lesquelles il pourrait établir une structure de m modules entreliés, en choisissant un nombre de variables variant de 1 jusqu'au maximum de $pm(m-1)/2$ connexions distinctes.

Ayant le choix de ne se servir d'aucune connexion (c'est-à-dire, spécifier la connexion nulle) le programmeur peut choisir entre l+1 variables distinctes (y compris la variable nulle inexistante). Le nombre maximal de structures internes possibles sera donc:

$$(l+1)^{pm(m-1)/2}$$

2) Connexions Externes: Le programmeur peut choisir parmi v variables distinctes servant de connexions d'entrée et de connexions de sortie. Le système a e variables d'entrée et s variables de sortie. Le programmeur peut choisir parmi v+1 variables distinctes (y compris la variable nulle inexistante). Le nombre maximal de structures externes possible sera donc:

$$(v+1)^{e}(v+1)^{s}$$

Nous montrons la distinction entre "interne" et "externe" en Figure II.5.5, où nous supposons que la bonne séquence algorithmique (étant LIRE suivie par ECRIRE) a été choisie:

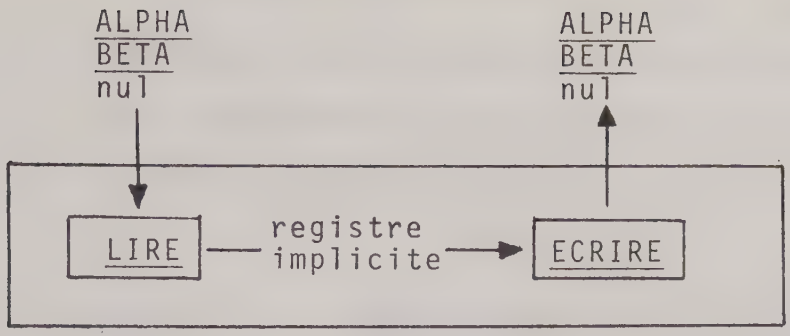

Figure II.5.5

Représentation relativiste d'un système ouvert devant représenter le module COPIER([BETA],[ALPHA]) vu de l'extérieur, et devant représenter l'algorithme LIRE(BETA), ECRIRE(ALPHA) vu de l'intérieur.

La formule combinatoire qui exprime le nombre maximal de structures que l'on peut synthétiser, dépendant du choix de variables servant de liens intermodulaires, est donc :

$$(l+1)^{pm(m-1)/2}(v+1)^{e}(v+1)^{s}$$

Dans le cas de notre exercice, $l=1$ (étant le registre interne implicite), $p=1$, $m=2$, $v=2$, $e=1$ et $s=1$. Etant donné que le registre interne implicite élimine --par le fait de son existence-- le choix de connexions internes, et que, par conséquent, une connexion nulle ne peut être

specifiée, le terme (l+1) se réduit dans ce cas à l. En supposant une séquence algorithmique correcte de LIRE suivie par ECRIRE, énumerons les $1^{1x2(2-1)/2}3^{1}3^{1} = 9$ structures possibles:

(1) LIRE(BETA)
ECRIRE(ALPHA)

(2) LIRE(BETA)
ECRIRE(BETA)

(3) LIRE(BETA)
ECRIRE( )

(4) LIRE(ALPHA)
ECRIRE(ALPHA)

(5) LIRE(ALPHA)
ECRIRE(BETA)

(6) LIRE(ALPHA)
ECRIRE( )

(7) LIRE( )
ECRIRE(ALPHA)

(8) LIRE( )
ECRIRE(BETA)

(9) LIRE( )
ECRIRE( )

Figure II.5.6

Enumération de $1^{pm(m-1)/2}(v+1)^{e}(v+1)^{s}$ structures.

Finalement, obtenons la formule qui exprime le nombre maximal de programmes que l'on puisse synthétiser à partir d'une enumération donnée de composants

$$M^{m}(l+1)^{pm(m-1)/2}(v+1)^{e}(v+1)^{s}$$

et notons également que cette formule ne comprend pas encore de facteur exprimant le choix de valeurs d'entrée pouvant être soit des valeurs valides soit des valeurs non valides. Selon cette formule, il y a 36 manières différentes de synthétiser notre système dont une seulement

--décrite en Figure II.5.6(1)-- est correcte! En allant de notre exercice irréel aux programmes de la vie réelle, nous trouvons des valeurs typiques telles que: $\underline{M}=256$, $\underline{m}=10^4$, $\underline{l}=10^5$, et $\underline{v}=10^5$.

TROISIEME PARTIE: TECHNOLOGIE

Nous avons défini une terminologie précise, nous permettant de parler d'un système informatique et de ses composants. Notre terminologie principale est limitée à quatre mots-clé:

- système
- module (ou, en terminologie plus générale "élément")
- variable (ou, en terminologie plus générale "lien", "connexion")
- valeur (ou, en terminologie plus générale "influence")

avec lesquels nous allons décrire l'approche méthodologique nous permettant de maîtriser les problèmes de la synthèse d'un système informatique. Nous distinguons entre la définition fonctionnelle du système (c'est-à-dire la conception et la décomposition modulaire du système) et l'implémentation d'une fonction donnée. Dans cette partie nous présentons notre approche technologique, en supposant que la définition fonctionnelle du système est donnée.

Nous allons décrire certains principes et certaines règles techniques, sans spécifier les détails des mécanismes pouvant les mettre en œuvre. Nous présentons notre technologie en termes abstraits, car premièrement, il y a plusieurs méthodes d'implémentation technique qui viennent à l'esprit et nous ne voulons pas donner la fausse impression que nous insistons sur une méthode plutôt que sur une autre; deuxièmement, notre intérêt théorique se repose sur notre pouvoir de décrire la technologie en termes systémologiques abstraits qui sont indépendants de toute connaissance informatique technique particulière; et troisièmement, notre technologie n'épuise pas tous les détails nécessaires à la fabrication réelle de software --la présenter de façon approfondie demanderait un ouvrage très volumineux, spécialisé au point de perdre de vue la perspective conceptuelle plus large-- par conséquent, elle ne couvre que suffisamment de détails pour que nous puissions démontrer l'application utilitaire de notre fondation théorique.

Cependant, afin de lier la description abstraite de cette technologie à des choses concrètes, nous ferons référence aux ouvrages de l'auteur, où de tels mécanismes pouvant mettre en œuvre notre technologie, sont présentés. Ces

références s'appliquent à la liste de nos ouvrages --représentant la source conceptuelle et expérimentale à partir de laquelle nous avons dérivé la théorie présentée-- donnée en appendice 1, indépendamment de la bibliographie générale.

III.1: LE PRINCIPE RELATIVISTE

Nous fondons notre technologie sur le principe inviolable de l'observation relativiste. Tandis que les autres principes, (aussi bien que les règles techniques) qui sont donnés plus loin, pourraient ne pas être respectés pour des cas exceptionnels où, dans la réalité, la nécessité de réaliser un produit particulier demande l'infraction partielle d'un principe ou d'une règle technique, nous exigeons que le principe relativiste ne soit jamais ignoré.

LE PRINCIPE RELATIVISTE: L'informaticien doit se considérer comme un observateur qui peut choisir sa perspective d'observation; il n'est capable que d'une observation interne du système de son choix, où le système est vu de l'intérieur en termes de modules reliés par des variables, et en termes de valeurs influençant le comportement fonctionnel de la structure modulaire observée. L'observateur ne peut jamais observer qu'un seul système à la fois.

C'est en insistant sur le principe relativiste, principe que nous considérons comme une loi absolue, que nous nous permettons, exceptionnellement, d'être tolérants

envers certaines infractions des autres principes et règles techniques; car, en respectant le principe relativiste, une telle infraction ne peut être que locale à un système observé particulier, et devient transparente à l'observateur, du point de vue de toute autre perspective d'observation. Ceci nous permet une flexibilité considérable d'adaptation aux réalités d'implémentation. Etant même donné un grand nombre d'infractions, l'informaticien professionnel, qui comprend le sens de notre technologie, saura toutes les rendre pratiquement invisibles à des niveaux d'observation, qui seraient soigneusement choisis comme niveaux modulaires essentiels. Du point de vue du synthétiseur du système, ces niveaux d'observation correspondent aux niveaux d'implementation*.

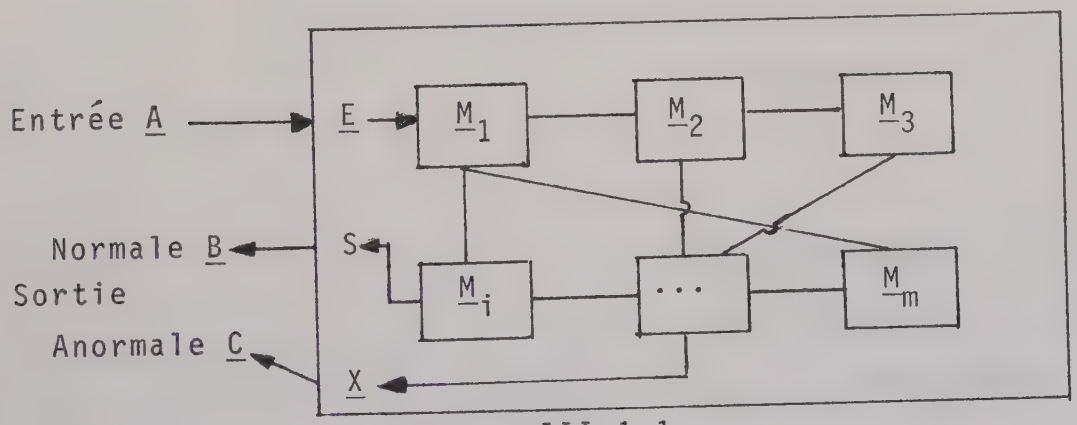

Figure III.1.1

Système ouvert observé F, représentant la fonction totale (b|c)=f(a), consistant de m modules structuraux M. Le système est vu uniquement de son intérieur! De ce point de vue local, les connexions externes A,B,C sont vues comme des variables locales E,S,X respectives.

* Voir "Niveaux d'implémentation" dans la section IV.1.

Le schéma en Figure III.1.1 représente le système ouvert en forme graphique. Nous attirons l'attention du lecteur sur le fait que l'observation est limitée à la localité qui est interne au système; les détails graphiques externes (A, B, C) représentent la manière par laquelle le module F est aperçu du point de vue d'un système englobant. Nous présentons en Figure III.1.2 le même système F en une notation informatique symbolique formelle. Ce formalisme sera expliqué dans les sections suivantes où nous ferons référence à la Figure III.1.2.

```
F: SYSTEME ([E1, E2, ..., Ee],[S1, S2, ..., Ss| X]);
MEMOIRES TEMPORAIRES(I1, I2, ..., Il);
     instruction 1;
                          ← point temporel ti
     instruction 2;
                          ← point temporel tj
     instruction 3;
        .
        .
        .
        .
     instruction m;
     RETOUR;
FINSYSTEME;
```

Figure III.1.2

Représentation du système F en notation informatique symbolique formelle. Les points temporels indiqués ne font pas partie du formalisme.

D'après la définition 3, un module est toujours un système. Rappelons au lecteur la notation informatique symbolique formelle représentant le système $\underline{F}$ du point du vue d'un système englobant pour lequel $\underline{F}$ est un module, faisant partie des $\underline{m}$ instructions du système englobant:

$$\underline{C} := \underline{F}([\underline{A}_1, \underline{A}_2, \ldots, \underline{A}_e], [\underline{B}_1, \underline{B}_2, \ldots, \underline{B}_s]);$$

III.2: CONCEPT DE LA LOCALITE

En Figure III.1.2 nous avons exprimé le système observé F par un formalisme symbolique de programmation. Nous indiquons, syntaxiquement, le début et la fin du texte représentant le système implémenté, en l'encadrant par les mots-crochets syntaxiques SYSTEME et FINSYSTEME. Le texte entre SYSTEME et FINSYSTEME représente l'implémentation de la fonction f; ce texte ne peut jamais contenir de mots-crochets SYSTEME/FINSYSTEME internes. Nous définissons "système inactif observé" comme étant la notation syntaxique de modules et de variables qui sont explicitement dénotés par le texte symbolique formel.* Notons que le nom de système F est externe au système! Ce nom n'est utile que si ce système sert de module pour un système englobant. Par le principe relativiste, le système ne s'aperçoit pas de son existence en tant qu'un objet nommé distinct, puisque d'après ce principe, le système observé est l'univers subjectif entier, ne contenant que des objets internes.

---

* Nous attirons l'attention du lecteur sur la différence entre le système inactif observé et le système actif réellement existant, tel qu'énoncé par le théorème 3 dans la section III.7.

Nous dirons qu'un objet (qui selon la définition 2 peut être un module ou une variable) est local au système observé si, et seulement si, son nom est explicitement dénoté par le texte. Nous définissons qu'un objet est interne au système observé s'il est un composant explicite ou implicite du système. Nous définissons comme objet interne implicite, tout objet interne qui n'est pas local au système. Nous déduisons qu'un objet interne implicite est le composant transparent d'un objet local, qui lui-même est toujours un objet interne explicite. Par élimination, l'univers externe au système observé comprend tout objet qui n'est pas interne.

Un programme est la représentation algorithmique d'une fonction de transformation de valeurs, exprimée en une séquence d'instructions (chacune de ces instructions étant un module) précisant une succession d'étapes de transformation de valeurs moindres. C'est en précisant la séquence exacte d'instructions, que nous spécifions, à l'ordinateur, l'ordre temporel désiré d'activations successives de ces instructions.

D'après la définition 10, l'ordre temporel spécifié est relatif à un déroulement de temps autonome. Du point de vue local du système observé, chaque instruction est

activée à un moment t qui est relatif au déroulement unilatéral de temps observé localement. Nous avons indiqué de tels points temporels t dans la Figure III.1.2; appelons le point temporel local t_i le moment d'invocation de l'instruction 2, et le point temporel local t_j le moment de retour de l'instruction 2; t_j est également le moment d'invocation de l'instruction 3.

D'après la définition 10, un système ouvert actif se désactive chaque fois qu'il invoque le nom d'un système à activer. Notre séquence d'instruction représente les noms de systèmes qui doivent être invoqués. En exécutant (terme informatique pour "invoquant le nom de") une instruction, le système se désactive; il est réactivé au moment de retour de cette instruction, pour se désactiver de nouveau en exécutant l'instruction suivante, etc. La durée d'activité d'un système se manifeste donc --du point de vue local du système actif observé-- par une séquence temporelle disjointe de périodes d'activités moindres. Nous sommes primordialement intéressés par deux points temporels précis dans l'activité observée d'un système, étant: premièrement le moment d'invocation du système observé, et deuxièmement le moment de retour du système observé. Définissons la période d'invocation du système observé comme étant la période --relative au déroulement de temps local-- s'étendant du

moment d'invocation jusqu'au moment de retour. Normalement, nous considérons la période d'invocation comme étant une seule continuité temporelle subjective.

Observation 4: Notons l'intérêt philosophique impliqué par la quantification du temps. Système F ne peut jamais être réellement actif, étant donné que toute activité consiste en l'invocation d'un module, ce qui a pour effet de soustraire le système observé au déroulement de temps. Le moment de retour du module activé n'est que le moment d'invocation du module succédant, et ainsi de suite. D'après le principe relativiste, tout module actif peut être choisi comme système observé; cette inactivité réelle s'applique donc, récursivement, à tout niveau d'observation, aussi moindre qu'il soit. Ultérieurement, on est ramené au niveau d'activité électronique.*

---

* Voir Spier Ref. 6, concernant l'aspect quantifié du déroulement de temps interne à l'ordinateur, et concernant la continuité subjective d'un déroulement de temps observé.

III.3: LA CONNAISSANCE LOCALE

Définissons le terme connaissance* --dans le cadre sémantique de cet ouvrage-- comme étant un ensemble de valeurs. Définissons que la connaissance soit dite locale si, et seulement si, elle ne comprend que les valeurs qui sont explicitement dénotées par le texte; autrement dit, la connaissance locale ne comprend que les valeurs qui ont un sens particulier puisque nous les avons explicitement identifiées dans le texte, au moment de la synthèse du système. L'observateur d'un système actif pourrait, évidemment, apercevoir d'autres valeurs qui influencent le comportement fonctionnel du système observé, mais qui ne sont pas des valeurs locales d'après notre définition (nous pourrions les appeler des valeurs internes transitoires).

Pour connaître un module, il nous faut savoir ses propriétés qualitatives représentant une fonction de transformation de valeurs $(\underline{b}|\underline{c})=\underline{f}(\underline{a})$, aussi bien que le nombre et la nature de ses connexions externes. Définissons la

---

* Le terme "connaissance" correspond au terme anglais technique "reference space" que nous avons choisi d'employer dans Spier Ref. 12, 13 et 14.

connaissance suffisante pour invoquer un module de manière fonctionnellement correcte, comme étant la somme de toutes les valeurs suivantes:

1) Le nom d'invocation F du module.

2) Le nombre e de variables d'entrée A, et la façon dont une valeur d'entrée a est distribuée parmi ces e variables.

3) Le nombre s de variables de sortie normale B, et la façon dont une valeur de sortie normale b est distribuée parmi ces s variables.

Définissons la connaissance totale d'un module comme étant la somme de la connaissance suffisante du module et de toutes les valeurs suivantes:

4) L'ensemble de valeurs constituant le domaine A de la fonction F.

5) L'ensemble de valeurs constituant le domaine transformé normal B de la fonction f.

6) L'ensemble de valeurs constituant le domaine transformé anormal C de la fonction f.

En faisant l'hypothèse que toutes les variables sont équivalentes, toute variable pouvant servir à la place de toute autre --une hypothèse qui nous convient dans le contexte de cet ouvrage, mais qui n'est pas réellement soutenable-- définissons que la connaissance suffisante d'une variable est son nom local*.

LE PRINCIPE DE CONNAISSANCE LOCALE: Un système ne doit jamais posséder qu'une connaissance locale de ses composants. La connaissance locale du système doit toujours être suffisante (ce qui n'exclut pas une connaissance locale plus étendue, soit même une connaissante locale totale). Par élimination, un système ignore (c'est-à-dire, est dépourvu de connaissance) tout objet qui ne lui est pas interne, aussi bien que tout objet interne implicite.

Etudions la séquence d'instructions donnée en la Figure III.1.2, où l'activité du système observé reprend aux points temporels $\underline{t}_i$ et $\underline{t}_j$. Au moment $\underline{t}_i$ l'instruction 2 est invoquée; du fait que l'activité du système observé reprend au moment $\underline{t}_j$, et d'après la définition 10, il s'ensuit que cette reprise d'activité du système observé a été accomplie par une invocation faite par l'intermédiaire

---

* Le nom local d'une variable n'est par nécessairement identique au nom de référence unique tel qu'énoncé par la définition 11.

du système actif qui nous est connu comme "instruction 2". Une telle invocation explicite serait en infraction du principe de connaissance locale, puisque cette invocation explicite exige une connaissance qui n'est pas locale à l'instruction 2. Pourtant, du fait d'avoir défini un comportement fonctionnel du système observé tel que décrit jusqu'à présent, il nous faut absolument un moyen d'invocation par lequel un module actif puisse réactiver son système englobant, qui ne lui est connu que par implication.

Règle Technique 1: Nous implémentons dans notre ordinateur une instruction de retour, de la forme syntaxique RETOUR (voir Figure II.1.2) qui a pour effet d'invoquer un système englobant qui ne nous est connu localement que par implication, de façon à ce que ledit système reprenne sa séquence d'activité à l'instruction succédant le module qui est le système actif présent.*

---

* Voir Spier Ref. 3, 5, 7, 8, 12 et 17.

III.4: COMPORTEMENT FONCTIONNEL PREDETERMINE

Définissons qu'un système actif a un comportement fonctionnel bien défini si, et seulement si, il se comporte comme une fonction bien définie. Dans ce cas nous disons que le comportement fonctionnel du système est prédéterminé, ce qui implique qu'il existe une connaissance totale de ce système représentant son comportement fonctionnel effectif (ce qui ne veut pas dire qu'une telle connaissance totale soit nécessairement locale à n'importe quel système observable).

Si le comportement fonctionnel d'un système n'est pas prédéterminé, nous disons qu'il est imprévisible, ce qui veut dire que le comportement fonctionnel du système n'est pas nécessairement incorrect, bien qu'il ne soit pas garanti d'être toujours correct. De manière itérative, nous pouvons définir le comportement fonctionnel correct comme étant celui correspondant exactement à une fonction bien définie dont il existe une connaissance totale.

Nous programmons notre algorithme en spécifiant une séquence d'instructions, où chaque instruction aura été soigneusement choisie en raison de sa propriété qualitative.

Si le comportement fonctionnel d'une instruction est imprévisible, il s'ensuit que le comportement fonctionnel du système est imprévisible. Par la définition 3, un module est toujours un système; par implication, tout système ouvert peut potentiellement servir de module structural.

Supposons un module donné G, qui est l'implémentation d'une fonction non totale d=g(h); supposons également que le comportement fonctionnel de G soit correct pour toute valeur d'entrée h qui est un élément du domaine H de la fonction g. Nous pouvons nous servir de G comme module structural d'un système synthétisé F, sans risquer que le comportement fonctionnel de F ne soit imprévisible, si, et seulement si, nous pouvons garantir le comportement fonctionnel prévisible de G par une vérification qui est externe a G. En ayant une connaissance totale de G (supposition tout à fait raisonnable), nous pourrions valider toute valeur d'entrée h de G. Malheureusement, le comportement fonctionnel d'un système peut également être affecté par des conditions circonstancielles relatives à son implémentation, et la validation de telles conditions est contraire au principe de connaissance locale, puisqu'elle nécessite une connaissance interne implicite.

Règle Technique 2: Un système F est toujours implémenté en forme de fonction totale (b|c)=f(a). L'implémentation d'un système F doit toujours inclure toute validation nécessaire (validation des valeurs d'entrée aussi bien que validation des conditions circonstancielles qui pourraient affecter le comportement fonctionnel de F), afin que le système F génère toujours une valeur de sortie correcte (que cette valeur soit de sortie normale ou de sortie anormale). La validation doit garantir également que le système F ait une période d'invocation finie.

III.5: ALLOCATION DYNAMIQUE DE MEMOIRE

Définissons une variable temporaire T comme étant une variable dont l'existence même est une fonction de la période d'invocation d'un système ouvert actif. L'existence de la variable est établie au moment d'invocation du système --par un moyen d'allocation de mémoire d'ordinateur--, et supprimée au moment du retour du système.* Définissons en outre qu'à l'instant de sa création**, la variable T est mise en un état particulier où elle ne contient aucune valeur. Nous stipulons que notre ordinateur n'est pas capable, par sa nature, d'effectuer l'observation d'une valeur inexistante. Si un système actif tente l'observation d'une variable contenant une valeur inexistante, l'ordinateur sera, par sa nature même, incapable d'effectuer une telle observation et se mettra dans un état d'arrêt. L'ordinateur n'est jamais incapable d'effectuer un rangement de valeur dans une variable, quel que soit la valeur redéfinie.

---

* Voir Spier Ref. 8, 9, 12 et 17 pour les détails techniques d'implémentation d'un mécanisme permettant l'allocation dynamique et la désallocation dynamique de mémoire d'ordinateur.

** Du fait de connaître le nom locat T avant la création de la variable, dont le nom de référence unique est à priori inconnu, il s'ensuit que l'identité entre le nom local et le nom de référence est une fonction du temps.

Règle Technique 3: Notre ordinateur doit posséder les mécanismes suivants:

1) Un mécanisme d'allocation dynamique de mémoire, à l'aide duquel il nous est possible de créer une variable temporaire I au moment d'invocation d'un système.

2) Un mécanisme de désallocation dynamique de mémoire, à l'aide duquel il nous est possible de supprimer l'existence d'une variable temporaire I au moment du retour d'un système.

3) Un mécanisme de mémoire comportant le concept d'une valeur inexistante.

4) Un mécanisme qui rend impossible l'observation d'une valeur inexistante; si une telle observation est tentée, l'ordinateur s'arrête.

Observation 5: Nous présentons notre technologie de manière abstraite. Des mécanismes correspondants aux points 1 et 2 du paragraphe précédent, existent, implémentés soit en hardware (voir Spier Ref. 9 et 12) soit en software (Spier Ref. 8 et 17). Autant que nous le sachions, des mécanismes hardware correspondant

aux points 3 et 4 n'existent pas. Leur utilité sera démontrée plus loin, dans la preuve du corollaire 1.2.

Règle Technique 4: A l'exclusion de ses connexions externes explicites (étant les connexions d'entrée E, les connexions de sortie normale S, et la connexion de sortie anormale X), un système ne peut avoir de connexions qui ne soient pas de variables temporaires T.*

D'après le principe de connaissance locale, toute variable T est une variable locale temporaire. Comme sera prouvé par le théorème 1, une variable T ne peut servir que de connexion interne au système, et correspond donc à une des l connexions internes discutées en section II.5.

Nous déclarons l'existence des l variables locales temporaires T par le formalisme suivant, tel que dénoté en Figure II.1.1:

MEMOIRE TEMPORAIRE(T_1, T_2, ..., T_l);

---

* Cette règle est théoriquement convenable, mais pratiquement intenable. Acceptons la pour le moment, car elle va nous faciliter les preuves de nos assertions. En fin de la partie III, nous parlons de la nécessité d'avoir des variables locales permanentes.

III.6: LIAISON EXTERNE DYNAMIQUE

Supposons un système englobant qui invoque le module F en spécifiant des connexions A, B et C. D'après le principe de connaissance locale, ces connexions sont locales au système englobant qui invoque F. Par la définition 3, tout module est un système. Le système invoqué F ne peut donc pas avoir connaissance de variables A, B et C du système englobant, puisqu'une telle connaissance serait en infraction du principe de connaissance locale.

Observation 6: En supposant que tout système soit toujours implémenté strictement conforme au principe de connaissance locale, il est évident qu'un tel système ne peut avoir de connexion commune avec aucun autre, puisque d'après la définition 11 une variable ne peut être atteinte que par référence à son nom, ce nom étant une valeur qui ne fait pas partie de la connaissance locale*. Nous faisons cette observation afin de montrer que la conformité rigoureuse à nos principes a pour effet de rendre le comportement

---

* Voir le théoreme 1 dans la section III.7.

fonctionnel de tout système semblable au comportement fonctionnel d'un système clos! Ceci est conforme à notre but énoncé, qui est de rendre autant que possible le système synthétisé semblable à un tas. La règle technique 5, qui suit, a pour but de rétablir la connectivité externe du système apparemment clos.

Pour qu'un système invoquant ("l'invoqueur") puisse échanger des valeurs avec le système invoqué, tout en étant conforme au principe de connaissance locale, nous stipulons l'existence d'un mécanisme de liaison dynamique qui établit, au moment de l'invocation, une identité entre les variables d'invocation (ou "arguments) A, B et C de l'invoqueur, et entre les connexions externes explicites (ou "paramètres") correspondants E, S et X de l'invoqué. De plus, étant donné que la connaissance suffisante de l'invoqué est locale à l'invoqueur aussi bien qu'a l'invoqué, et par conséquent doit être connue du mécanisme de liaison dynamique, ce mécanisme serait en mesure de vérifier la validité de cette connexion externe parmi deux systèmes qui sont effectivement hors d'atteinte l'un de l'autre*.

---

* Voir Spier Ref 1, 3, 5, 7, 8, 9, 13 et 17.

Règle Technique 5: Notre ordinateur est de nature à posséder un mécanisme de liaison dynamique, établissant une identité entre les arguments (A, B et C) de l'invoqueur, et les paramètres correspondants (E, S et X) de l'invoqué*. Cette identité est établie dynamiquement au moment de l'invocation, et reste en vigueur pendant la période de cette invocation. Cette identité est dynamiquement supprimée au moment du retour du système invoqué. Ce mécanisme de liaison dynamique vérifie la correspondance exacte entre les éléments des ensembles (A et E), (B et S) et (C et X), aussi bien que la distribution de valeurs parmi ces variables, conforme à la connaissance suffisante du système invoqué. Toute connexion non valide aurait pour effet de mettre l'ordinateur en état d'arrêt. La liaison dynamique est établie de façon telle que la connexion d'entrée (A et E) ne puisse être qu'observée et que les connexions de sortie (B et X) et (C et X) ne puissent être que redéfinies. Le mécanisme de liaison dynamique exige, en plus, que la connexion d'entrée soit observée et que les connexions de sortie soient redéfinies. Toute infraction, qu'elle soit redéfinition de l'entrée,

---

* Par la nature de la variable temporaire, il existe un mécanisme similaire pour établir (et rompre) une identité entre le nom local T et le nom de référence de la variable elle-même.

observation de la sortie, manque d'observation de l'entrée, ou manque de redéfinition de la sortie, aurait pour effet de mettre l'ordinateur en état d'arrêt.

Comme démontré en Figure III.1.1, nous déclarons les connexions externes du système F par le formalisme:

F: SYSTEME([$\underline{E}_1$, $\underline{E}_2$, ..., $\underline{E}_e$],[$\underline{S}_1$, $\underline{S}_2$, ..., $\underline{S}_s$ | $\underline{X}$]);

où les variables E, S et X sont dites paramètres formels puisqu'elles correspondent non pas à des variables spécifiques, mais aux identités implicites des arguments A, B et C correspondants qui ont été spécifiés par l'invoqueur.

III.7: PROPRIETES D'UN SYSTEME IMPLEMENTE

En supposant que tout système soit implémenté selon nos règles techniques, nous pouvons faire des assertions concernant ses propriétés.

Théorème 1: Une variable locale temporaire T, qui est le composant local d'un système implémenté F, ne peut jamais servir de connexion externe liant ce système F à son univers externe.

Preuve: Avant le moment d'invocation de F, la variable T est inaccessible dans l'univers externe à F puisqu'elle n'existe pas encore. Suivant le moment du retour de F, la variable T est inaccessible dans l'univers externe à F puisqu'elle n'existe plus. En vertu de la connaissance locale du système F, ce système ne peut explicitement invoquer que ses modules locaux; donc, pendant la durée de son existence, la variable T est inaccessible dans l'univers externe à F puisque pendant la période d'invocation de F toute activité est --en vertu de la connaissance locale-- interne à F. Par élimination, tout système externe à F est inactif, et, d'après la définition 5, un système inactif

se comporte comme s'il était un système clos; par définition, un système clos se comporte comme s'il n'avait aucune connexion externe.

Corollaire 1.1: Le comportement fonctionnel d'un système ouvert actif F ne peut être influencé par son univers externe qu'à l'aide de ses connexions externes explicites E, S et X.

Preuve: A l'exclusion de ses paramètres formels, qui sont les connexions externes de F, le système ne comprend que des variables locales temporaires T. D'après la définition 1, une valeur ne peut être transmise qu'à l'aide d'une variable. D'après le théorème 1, une variable locale temporaire T ne peut jamais servir de connexion externe.

Corollaire 1.2: Le comportement fonctionnel du système F ne peut être influencé que par des valeurs provenant de (c'est-à-dire ayant leur origine dans) l'univers externe a F.

Preuve: D'après le corollaire 1.1, le système ne peut être influencé par son univers externe qu'à l'aide de ses

paramètres formels. Toute autre influence interne serait une influence de sortie provenant des modules locaux qui ont un comportement fonctionnel prédéterminé, une telle influence de sortie étant dérivée de l'influence provenant de l'univers externe à F. La seule influence imprévisible ne provenant pas de l'univers externe à F serait une influence venant d'une variable locale temporaire I, et d'après la règle technique 3, il est impossible à une variable I de contenir une valeur observable ne provenant pas de l'univers externe à F.

Théorème 2: Le système F ne peut être influencé par son univers externe que par sa valeur d'entrée explicite a. Il ne peut influencer son univers externe que par ses valeurs de sortie explicites b et c. Le système F est toujours influencé par sa valeur d'entrée explicite a. Le système F influence toujours son univers externe par ses valeurs de sortie explicites b et c.

Preuve: D'après le corollaire 1.1, le système ne peut être influencé par son univers externe qu'à l'aide de ses connexions externes explicites E, S et X. Par la nature du mécanisme de liaison dynamique, tel qu'énoncé

dans la règle technique 5, le système F ne peut qu'observer sa valeur d'entrée a et est incapable d'observer la valeur de S ou de X. Egalement, le système F ne peut que redéfinir la valeur de S et de X, et est incapable de redéfinir la valeur de E.

Théorème 3: Un système F n'existe réellement que pendant sa période d'invocation.

Preuve: Le système F ne contient que deux genres de variables: des paramètres formels qui ne correspondent à aucune variable spécifique, et des variables locales temporaires T. D'après la définition 5, un système structuré consiste en modules entreliés par des variables. Il s'ensuit que le système F ne peut pas réellement exister si ses variables n'existent pas. Or, les paramètres formels de F n'ont d'identité réelle que pendant la période d'invocation de F, et les variables T n'ont d'existence que pendant la période d'invocation de F. Donc, les variables locales à F n'ont d'existence réelle que pendant sa période d'invocation.

Corollaire 3.1: Toute invocation d'une implémentation F de la fonction f résulte en la création d'un système réel unique F, qui est distinct de tout autre système actif.

Preuve: D'après la définition 11, une variable est un objet discret, ayant une identité unique. Par conséquent, toute invocation de F est un système réel composé de variables qui sont uniques à ce système, donc il est distinct de tout autre système actif.

Corollaire 3.2: L'implémentation correcte F d'une fonction f peut être invoquée récursivement sans que le comportement fonctionnel correct de F ne soit affecté.

Preuve: D'après le corollaire 3.1, toute invocation de F résulte en la création d'un système réel distinct et unique; l'invocation récursive de F résulterait en l'existence d'autant de systèmes réels distincts et uniques qu'il y avait d'invocations de F. Etant des systèmes distincts, ils ne peuvent s'influencer que par leurs connexions externes formelles qui exhibent toujours le comportement fonctionnel correct de F, puisque par l'énoncé du corollaire 3.2 l'implémentation de F est correcte.

Comme décrit dans l'introduction de cette partie, la technologie proposée n'épuise pas tous les détails techniques nécessaires à la production réelle de software. De manière

analogue, nous n'avons pas nécessairement épuisé toutes les assertions qui peuvent être faites sur les systèmes implémentés selon nos règles techniques. Notre intérêt théorique repose sur le fait qu'à l'aide de treize définitions, deux principes et cinq règles techniques, il nous est possible de parler de manière très précise d'une partie non triviale de l'informatique, sans devoir s'appuyer sur des connaissances techniques spécialisées, ni devoir se servir de jargon informatique particulier, ni devoir se servir d'exemples. Nous estimons avoir démontré que notre approche systémologique pourrait être d'utilité théorique étendue.

III.8: IMPLEMENTATION ALGORITHMIQUE

Nous arrivons enfin aux règles qui nous permettraient de synthétiser les représentations algorithmiques correctes de nos fonctions définies. En supposant que tout système ne soit implémenté que conformément à nos règles techniques, parlons de l'implémentation. Les règles techniques 1, 3, 4 et 5 ne nécessitent pas la compréhension du problème; par contre, la règle technique 2 n'est réalisable qu'à condition que d'une part, nous comprenions notre problème et sachions l'implémenter, et que d'autre part nous pouvions garantir le comportement fonctionnel correct du système implémenté.

Nous exigeons donc que le système soit simple, sinon il serait incompréhensible. Nous estimons, à titre de postulat fondé sur de nombreuses observations dans les milieux de la programmation, qu'un programme à volume de détail limité (supposons entre 50 et 100 instructions, et entre 10 et 20 variables) sera toujours intellectuellement assimilable par le cerveau du programmeur moyen, et qu'il constituera donc par définition un système informatique simple.

Règle Technique 6: Le volume de détail représenté par le texte du programme ne doit jamais dépasser le seuil au delà duquel il ne serait plus intellectuellement assimilable par le cerveau du programmeur professionnel moyen.

Cette règle s'applique litéralement au nombre d'instructions du programme, peu importe que ce programme soit écrit en langage de machine hardware ou en langage de programmation de haut niveau. Une instruction d'un langage de programmation de haut niveau est, en général, fonctionnellement plus puissante qu'une instruction de machine hardware. Par conséquent, un même nombre d'instructions d'un langage plus puissant peut représenter un contenu fonctionnel plus riche que celui d'un langage moins puissant. Pourtant, quel que soit le langage de programmation, il y existe toujours ce seuil de volume de détails d'instructions, au delà duquel on excède le pouvoir d'assimilation intellectuelle d'information, propre au cerveau humain.

Même en suivant rigoureusement la règle technique 6, nous n'avons pas garanti que l'implémentation du programme

est réellement correcte; nous avons seulement garanti que le programme n'est pas incompréhensible! Etant donné que c'est le comportement fonctionnel du programme --qui est sa propriété qualitative de la forme $(\underline{b}|\underline{c})=\underline{f}(\underline{a})$-- qui peut être incompréhensible en termes des détails composants énumérés (qui sont le texte du programme), admettons notre impuissance à valider ce comportement fonctionnel par des moyens intellectuels! Laissons le système implémenté nous montrer son comportement fonctionnel; puisque ce comportement est --par la définition 12-- une propriété observable, observons-la! Si elle est apparemment satisfaisante, nous pouvons alors déclarer que le système est acceptablement correct; sinon, la compréhensibilité de l'algorithme (qui nous est garantie par la règle technique 6) nous permettra de trouver l'erreur et de la corriger.

Supposons que nous ayons un programme vérificateur existant, connu comme ayant un comportement fonctionnel correct, qui nous permet d'invoquer le programme implémenté $\underline{F}$ de manière telle que le programmeur puisse fournir à $\underline{F}$ des valeurs d'entrée $\underline{a}$ valides aussi bien que des valeurs d'entrée $\underline{x}$ erronées, en fournissant à $\underline{F}$ une valeur à la fois. Après chaque invocation de $\underline{F}$, le programme

vérificateur restitue au programmeur les valeurs de sortie générées par F, étant la valeur de sortie normale b et la valeur de sortie anormale c. Cette vérification est interactive, où les valeurs d'entrée sont fournies par le programmeur à l'aide d'un pupitre de contrôle, et où les valeurs de sortie sont restituées au programmeur à l'aide du même pupitre. Le programmeur est effectivement l'observateur qui vérifie le comportement fonctionnel du programme par expérimentation. Les preuves données dans la section III.7 nous garantissent l'invariance du comportement fonctionnel observé.

Règle Technique 7: La vérification du comportement fonctionnel du système synthétisé fait partie intégrale du processus de synthèse. Si le comportement fonctionnel observé est erroné, l'erreur doit être corrigée et le système doit être soumis à une vérification nouvelle. Ce processus itératif doit être suivi, jusqu'à ce que le comportement fonctionnel du programme soit apparemment satisfaisant. Le programme est alors déclaré être acceptablement correct.

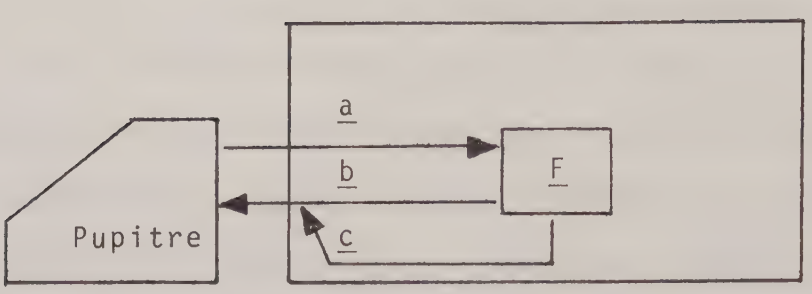

Figure III.8.1

Schéma du programme vérificateur pour effectuer l'observation interactive du comportement fonctionnel du module F.

Observation 7: Nous ne prétendons pas que les systèmes synthétisés en suivant nos règles techniques sont rigoureusement corrects. Nous avons décrit ces systèmes comme étant "d'un comportement fonctionnel apparemment satisfaisant".* Cependant, nous pouvons garantir que tant que le système vérifié n'est pas explicitement modifié par programmation (cas auquel il doit être revérifié, selon la règle technique 7), son comportement fonctionnel observé restera invariant. Nous pouvons donc synthétiser et vérifier un seul système à la fois, ce qui est conforme à notre principe relativiste inviolable; puis, nous pouvons nous servir de ces systèmes vérifiés comme modules qui serviraient à la synthèse d'un système englobant plus important, en ayant la garantie du comportement

---

* Voir Spier Ref. 13.

fonctionnel invariant de tout système vérifié. Nous avons appliqué cette technologie à la fabrication de software industriel, et avons observé une réduction spectaculaire (très souvent même la réduction totale) du nombre d'erreurs dans nos programmes; cet effet est généralement accompagné d'une augmentation mesurable de la productivité du programmeur. De notre point de vue pragmatique, nous n'insistons donc pas sur des preuves formelles et rigoureuses de la validité de nos programmes, et trouvons la probabilité d'un très petit nombre d'erreurs résiduelles tolérable.

Finalement, nous admettons n'avoir pas parlé de variables locales d'allocation de mémoire permanente, sans lesquelles il est pratiquement impossible de programmer quoi que ce soit. Ces variables seraient déclarées par le formalisme:

MEMOIRE PERMANENTE($\underline{P}_1$, $\underline{P}_2$, ..., $\underline{P}_k$);

Les avoir présentées de manière rigoureuse aurait pratiquement doublé le volume de la troisième partie. Le sujet a été traité dans Spier, Ref. 9, 12 et 21 de manière

suffisamment détaillee pour que le lecteur soit assuré que nous n'avons pas ignoré cet aspect de la programmation. En bref, il est bien possible de programmer un système qui soit le gérant exclusif d'une base de données rangée en mémoire locale permanente (nous dénotons un tel système dans Spier Ref. 3, 5, 9 et 12, par le terme anglais "caretaker procedure"), et qui se présente, du point de vue externe de son comportement fonctionnel observable, comme tout système implémenté selon nos sept règles techniques. Le lecteur intéressé trouvera dans Spier Ref. 9 et 12 une description technique extrêmement détaillée sur l'implémentation possible d'un tel système.

III.9: REDUCTION DE LA COMPLEXITE

Dans la section II.5 nous avons présenté une formule exponentielle exprimant le nombre maximal de systèmes qui peuvent être synthétisés à partir d'une énumération donnée de composants

$$\underline{M}^{\underline{m}}(\underline{l}+1)^{\underline{pm}(\underline{m}-1)/2}(\underline{v}+1)^{\underline{e}}(\underline{v}+1)^{\underline{s}}$$

où les termes $\underline{l}$ et $\underline{v}$ représentent le choix de variables servant de liens intermodulaires.

Le facteur $\underline{M}^{\underline{m}}(\underline{l}+1)^{\underline{pm}(\underline{m}-1)/2}$ représente le nombre maximal de structures pouvant être synthétisées à l'intérieur du système. En section III.8 nous avons admis notre impuissance à comprendre un système structuré à partir de l'énumération de ses détails; nous avons également insisté sur la validation du comportement fonctionnel du système par expérimentation et observation, en stipulant que le système soit rendu compréhensible en vertu de sa taille limitée. Cette compréhensibilité nous permet de trouver et corriger toute erreur observée. Par conséquent, peu importe lequel des systèmes possibles a été synthétisé, à condition qu'il ait un comportement fonctionnel apparemment satisfaisant. La différence entre ce système et tout autre ayant un comportement fonctionnel identique ne peut être que

quantitative. Il s'ensuit qu'en vertu des règles techniques 6 et 7, ce nombre énorme de possibilités ne doit plus nous effrayer, puisque nous pouvons atteindre un résultat satisfaisant sans devoir chercher à comprendre pourquoi.

Le facteur $(\underline{v}+\underline{1})^{\underline{e}}(\underline{v}+\underline{1})^{\underline{s}}$ représente le choix de variables servant de connexions externes, et ce choix n'existe plus en vertu des règles techniques 5, 6 et 7.

QUATRIEME PARTIE: CONCEPTION

Tandis que la fabrication d'un système préconçu est un métier pratiqué par l'artisan, la conception de ce système est un art. C'est un fait bien connu que les anciens maîtres de la peinture (ou de la sculpture) ont souvent délégué à leurs apprentis la réalisation technique de leurs œuvres. Au même titre, la conception d'un immeuble ou d'un navire est un art qui est pratiqué par l'architecte. Ces systèmes sont alors réalisés par des techniciens, conformément aux spécifications détaillées de l'architecte. L'architecture peut s'apprendre à l'école; la qualité fonctionnelle et esthétique du système conçu par l'architecte dépend du talent artistique et créateur dont cet architecte est doué.

Ceci s'applique également à la conception d'un système informatique. A présent, il est courant de parler de l'architecture d'un système informatique. Dans le métier de la programmation, on distingue entre l'architecte du système (la désignation professionnelle anglaise étant "system architect"), et le programmeur qui est chargé de l'implémentation d'un système décomposé en modules bien définis*.

---

* Voir Mills (Ref. 38, 39 et 40).

Bien que fréquemment ce soit l'architecte du système qui se charge aussi de l'implémentation (ou d'une partie de l'implémentation), nous devons distinguer entre ces deux rôles, même s'ils sont joués par la même personne.

L'implémentation d'une fonction donnée, lorsqu'effectuée de manière méthodique en suivant les règles techniques données dans la partie précédante, résultera en un programme vérifié ayant un comportement fonctionnel prévisible. La conception --couronnée de succès-- du système dépend largement du don créateur de l'architecte. Par conséquent, nous ne pouvons que présenter nos règles pour la décomposition modulaire d'un système, sans cependant, pouvoir garantir la qualité du résultat. Nous présentons une approche qui permettrait à l'architecte doué de rendre le système conçu très insensible aux perturbations, en tenant compte par avance du fait que de telles perturbations seront inévitables. Les perturbation dont nous parlons sont les modifications --par programmation-- d'un système existant, nécessaires pour les besoins des usagers.

Prenons par exemple le programme "calcul de paye", et supposons qu'il ait été implémenté correctement. D'après

la définition 6, toute perturbation (par programmation) de ce système existant aura pour effet de redéfinir la propriété qualitative du système. Une fois qu'un système a le comportement fonctionnel désiré, il s'ensuit que nous ne devrions, en principe, plus jamais modifier quoi que ce soit dans ce système. Pourtant, il se pourrait par exemple, que le gouvernement ait décrété des lois nouvelles, nécessitant un changement du processus du calcul de paye. Le gouvernement ne s'intéresse point à la manière par laquelle nous avons choisi d'effectuer le calcul; que cela soit le travail manuel d'un comptable ou le travail automatique d'un calculateur électronique, ne pas faire le calcul tel que décrété serait en infraction à la loi. Que cela nous plaise ou non, nous nous trouvons dans l'obligation de reprogrammer certaines parties de notre système. En prévoyant cette éventualité, le bon architecte devrait pouvoir organiser son système de manière à réduire au maximum les effets indésirables d'une telle perturbation.

IV.1: NIVEAUX D'IMPLEMENTATION

Nous specifions l'algorithme de notre programme par une séquence d'instructions, chaque instruction pouvant être un programme moindre, composé à son tour d'instructions qui peuvent être des programmes encore moindres, etc. Ultimement, nous nous servons d'instructions faisant partie de l'ordinateur hardware lui-même. Appelons nos instructions fonctions génératrices, puisqu'elles nous permettent la génération d'une entité fonctionnelle plus importante.

Dans la section II.5, nous avons dit que le système était composé de m modules, chacun ayant une, parmi M propriétés qualitatives distinctes données. Définissons l'ensemble générateur comme étant un ensemble de M fonctions génératrices G distinctes (G_1, G_2, ..., G_M). Définissons l'ensemble générateur comme étant suffisant s'il nous est possible de synthétiser un système F à partir de cet ensemble d'instructions G_i.

Evidemment, l'ensemble générateur suffisant initial est l'ensemble des instructions machine hardware. Nous ne pouvons implémenter un programme que s'il est ultimement exprimable en termes d'instructions machine hardware.

Nous pouvons, cependant, nous imaginer un ensemble générateur suffisant, où chaque instruction de l'ensemble est implémentée en software, par programmation.

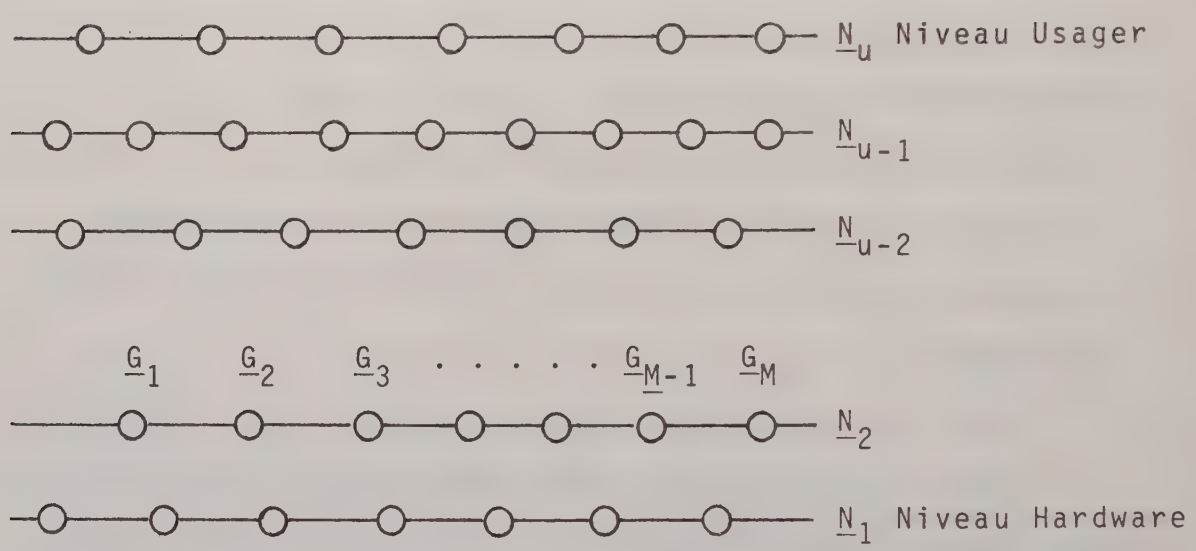

Figure IV.1.1

Représentation de u niveaux d'implémentation. Les lignes horizontales représentent le niveau, les cercles représentent les M fonctions génératrices G implémentées à ce niveau.

Imaginons donc une hiérarchie de u niveaux d'implémentation, telle qu'illustrée en Figure IV.1.1, où chaque niveau N_i consiste en M fonctions génératrices G distinctes, implémentées conformément à nos règles techniques de la section précédente, à l'exclusion du niveau N_1 qui consiste

en M modules réalisés en hardware. Nous faisons la supposition qu'un module réalisé en hardware se comporte exactement comme s'il avait été implémenté conformément à nos règles techniques.

Tout niveau N_i est appelé machine, où N_1 est une machine hardware, et N_2 jusqu'à N_u sont des machines software. Nous établissons la règle par laquelle l'implémentation d'un système, à un niveau donné N_i ne peut être effectuée qu'en utilisant des instructions implémentées aux niveaux N_1 jusqu'à N_{i-1}. Il s'ensuit que nous pourrions toujours éliminer les niveaux d'implémentation N_i jusqu'à N_u sans affecter le comportement fonctionnel établi des systèmes existant aux niveaux d'implémentation N_1 jusqu'à N_{i-1}. Ceci dit, il devient clair qu'en implémentant aux niveaux les plus élevés tout système pouvant être assujetti à des modifications de nature imprévisible, nous pourrons délimiter les effets d'une telle modifications. En supposant la modification d'un système F, existant au niveau d'implémentation N_j, toute perturbation potentielle résultant de la modification de F sera limitée à F même, et à tout système existant aux nivaux d'implémentation N_{j+1} jusqu'à N_u.

En raison de nos règles techniques d'implémentation, tout système existant à un niveau donné N_i se comporte comme un système clos envers tout autre système du même niveau

d'implémentation. Chacune de nos machines ("machine" étant l'ensemble d'instructions données à un certain niveau d'implémentation) peut donc être décrite, d'après la définition 5, comme étant un tas d'instructions. Ceci est d'une importance capitale, étant donné qu'un tas n'a que des propriétés quantitatives. On peut donc toujours ajouter des objets à un tas, sans qu'il soit affecté autrement que de manière quantitative (c'est-à-dire en devenant plus grand).

Bien que la modification du système $\underline{F}$ qui existe au niveau d'implémentation $\underline{N}_j$ n'affecte pas les niveaux d'implémentation $\underline{N}_1$ jusqu'à $\underline{N}_j$ (à l'exclusion de $\underline{F}$), il n'est pas garanti que la modification de $\underline{F}$ pourra effectivement être accomplie, car il est possible que cette modification nécessite une fonction génératrice $\underline{G}_{M+1}$ du niveau $\underline{N}_i$ (où $i < j$), et qu'une telle instruction n'y existe pas. Or, d'après la propriété-tas du niveau $\underline{N}_i$, il est garanti que nous pouvons toujours ajouter une instruction de plus à ce niveau, sans affecter le comportement fonctionnel d'aucun autre système existant; par conséquent, nous pouvons garantir que la modification de $\underline{F}$ sera possible (à condition, toujours, que le nouvel algorithme $\underline{F}$ soit ultimement exprimable par l'ensemble générateur du niveau $\underline{N}_1$).

IV.2: DECOMPOSITION MODULAIRE

Dans la section I.3, nous avons identifié trois raisons qui contribuent à l'incompréhensibilité d'un problème. Ces raisons étaient:

- Manque de définition
- Volume
- Irrationalité

Nos règles techniques d'implémentation garantissent la rationalité, dans le sens que chaque programme se comporte --à l'exception de ses connexions externes-- comme s'il était un système clos. Par conséquent, le comportement fonctionnel de tout programme est essentiellement indépendant de celui de tout autre. La règle technique 6 garantit la compréhensibilité d'un programme bien défini, en imposant une limite sur la quantité des composants de ce programme.

Dans la partie II.1 nous avons dit qu'en général, l'énoncé du problème informatique, non mathématique, est charactérisé par son manque de définition. Par exemple, si l'énoncé du problème est: "concevoir un système de gestion pour des applications comptables, destiné à l'usage général des banques", le problème est *à priori* insoluble,

étant donné que chaque banque a ses pratiques de comptabilité particulières, qui peuvent bien être de nature incompatible avec les pratiques comptables d'une autre banque. Ces particularités ne doivent être évidentes qu'au niveau de l'usager, qui est le niveau de l'implémentation $\underline{N}_u$ en Figure IV.1.1. Nous suggérons que le problème pourrait être rendu compréhensible, si nous pouvions rendre le niveau d'implémentation $\underline{N}_{u-1}$ bien défini, en ignorant temporairement le niveau $\underline{N}_u$.

Ceci peut se faire, si nous arrivons à bien définir, non pas la totalité d'applications d'usagers spécifiques, mais la classe de telles applications que notre système pourra éventuellement supporter. C'est ici qu'entre en jeu le facteur de l'intelligence et de la créativité de l'architecte. C'est à lui de définir ce qu'est une classe d'applications. Une fois cette classe définie, l'architecte doit être pourvu de facultés analytiques, le permettant de dériver l'invariance fonctionnelle qui est commune à toutes les applications faisant partie de cette classe.* Ayant dérivé cette invariance fonctionnelle, l'architecte aura temporairement exclu de sa considération toutes les singularités relatives aux applications banquaires particulières.

---

* Voir Spier Ref. 16, où nous décrivons --en grand détail-- la conception d'une invariance fonctionnelle qui est commune à toute une classe d'applications, ainsi que ses détails d'implémentation.

Cette invariance fonctionnelle peut maintenant être exprimée par un ensemble générateur suffisant qui constituera le niveau d'implémentation $\underline{N}_{u-1}$. Chacune de ces instructions pourrait encore être de volume excessif, et par conséquent inimplémentable (selon la règle technique 6). L'architecte doit alors dériver l'invariance fonctionnelle qui est commune à toutes les instructions de la machine $\underline{N}_{u-1}$, afin de définir l'ensemble générateur $\underline{N}_{u-2}$ qui permettra l'implémentation --conformément à la règle technique 6-- de tout système faisant partie du niveau $\underline{N}_{u-1}$. L'architecte doit répéter ce processus de décomposition modulaire jusqu'au niveau $\underline{N}_2$ qui est implémentable --conformément à la règle technique 6-- en instructions machine hardware.

Ce processus de décomposition modulaire, par dérivation d'une invariance fonctionnelle qui est commune à toutes une classe de fonctions, se fait rarement sans erreurs. Typiquement, en étant arrivé à un certain niveau de décomposition modulaire $\underline{N}_i$, l'architecte s'aperçoit qu'en ayant pris des décisions différentes à un niveau plus élevé $\underline{N}_j$ (où $\underline{u} > \underline{J} > \underline{i}$), il aurait pu atteindre une décomposition modulaire plus simple, plus compréhensible, ou peut-être plus esthétique. Dans ce cas-là, il remonte

au niveau $\underline{N}_j$, redéfinit l'ensemble générateur à ce niveau, et recommence la décomposition molulaire à partir de ce niveau.

Quand le processus de décomposition modulaire est accompli de manière satisfaisante, on implémente le système en remontant les niveaux. Tout d'abord on implémente, indépendemment, chaque système faisant partie du niveau $\underline{N}_2$, en suivant rigoureusement les règles techniques d'implémentation. Puis on se sert de la machine $\underline{N}_2$, qui est maintenant une machine existante, pour implémenter, indépendemment, chaque système faisant partie du niveau $\underline{N}_3$, etc. jusqu'au niveau $\underline{N}_{u-1}$. Nous avons alors un système qui est encore incapable de gérer les applications comptables d'une banque, mais qui est capable d'exprimer --en raison de sa nature, étant un ensemble générateur suffisant-- tout algorithme de gestion comptable faisant partie de la classe d'applications d'usagers pour laquelle il a été conçu.

Il est maintenant facile de programmer chaque application banquaire individuelle, en se servant des fonctions génératrices existantes. Ce qui nous ramène, finalement, au probleme initial qui était: "comment empreindre à

l'ordinateur électronique le comportement fonctionnel exigé par l'usager humain?".

Nous avons effectué un nombre non négligeable de conceptions et d'implémentations réelles de systèmes, en suivant les règles données dans cette thèse. Nous avons établi, par expérimentation, que si l'on sait isoler les constructions logiques --correspondant aux exigences de l'usager humain-- au niveau $\underline{N}_u$ de l'implémentation de l'ultime système, le volume de détail (étant le nombre d'instructions) correspondant à ce niveau, ne représente, typiquement, qu'entre 5% et 25% du volume de détail de l'implémentation entière. Si l'implémentation du niveau $\underline{N}_u$ se trouve être encore trop volumineuse --et par conséquent incompréhensible-- l'architecte doué saura toujours décomposer le niveau $\underline{N}_u$ en sous-niveaux de taille plus maniable.

APPENDICE 1: LISTE DES OUVRAGES DE L'AUTEUR

Nous présentons ci-dessous la liste des ouvrages publiés par l'auteur, en témoignage de ses recherches effectuées dans des milieux universitaires et industriels. Ces recherches, dont certaines sont de nature abstraite et d'autres de nature pragmatique (expériences basées sur les idées de l'auteur), touchent à différents aspects de l'ouvrage présent.

Cet ouvrage dérive de ces recherches. Il représente nos conclusions philosophiques et théoriques dans le cadre unificateur de la systémologie.

(1) Spier M J, Organick E I, The Multics Interprocess Communications Facility, Proc. 2nd ACM SIGOPS Symposium on Operating Systems Principles, Princeton NJ, October 1969, pp. 83-91.

(2) Spier M J, Organick E I, An Operating System Model Featuring Demand Scheduling of Virtual Resources, Proc. 3rd Hawaii International Conference on System Sciences, Honolulu, January 1970, pp. 704-707.

(3) Spier M J, A Computer System Model for Controlled Sharing of Information, Proc. Online72, Brunel University, Uxbridge England, September 1972, pp. 243-263.

(4) Spier M J, Towards an Operating System Oriented Hardware, IEEE Workshop on Computer Architecture and Operating Systems, University of Austin, January 1973 (Working paper & presentation notes; unpublished).

(5) Spier M J, A Model Implementation for Protective Domains, International Journal for Computer and Information Sciences vol. 2 #3 (1973), Plenum Press NY, pp. 201-229.

(6) Spier M J, Process Communication Prerequisites, or the IPC-Setup Revisited, Proc. 1973 Sagamore Computer Conference on Parallel Processing, University of Syracuse NY, August 1973, pp. 79-88.

(7) Spier M J, The Experimental Implementation of a Comprehensive Intermodule Communications Mechanism, Proc. 1973 Sagamore Conference on Parallel Processing, University of Syracuse NY, August 1973, pp. 89.

(8) Spier M J, Macro-implemented Structured Programming Mechanism, First Tuesday Computer Science Lecture, Digital Equipment Corporation, September 1973, Internal Publication.

(9) Spier M J, Hasting T N, Cutler D N, An Experimental Implementation of the Kernel/Domain Architecture, Proc. 4th ACM Symposium on Operating System Principles, Yorktown Hights NY, October 1973, pp. 8-21.

(10) Spier M J, PDQ - A Formal Text Editing Language, Digital Equipment Corporation, February 1973, Internal Publication.

(11) Spier M J, Some Principles Worth Remembering in Operating System Design, Invited Lecture and Presentation notes, Operating System Festival, University of Massachusetts, Amherst MA, February 4-5 1974. Unpublished.

(12) Spier M J, Hastings T N, Cutler D N, A Storage Mapping Technique for the Implementation of Protective Domains, Software - Practice & Experience, vol. 4 #3, 1974, pp. 215-230.

(13) Spier M J, A Pragmatic Proposal for the Improvement of Software Modularity and Reliability, International Journal of Computer and Information Sciences, vol. 4 #2, 1975.

(14) Spier M J, A Vernacular Presentation of the Domain Machine, in preparation.

(15) Spier M J, A Critical Look at the State of our Science, ACM SIGOPS Operating Review, vol. 8 #2 (1974), pp. 9-15.

(16) Spier M J, Stein T J, Bricklin D, The Typeset10 Message Exchange Facility - A Case Study in Systemic Design, Proc. 1974 Sagamore Conference on Parallel Processing, University of Syracuse NY, August 1974.

(17) Spier M J, The Typeset10 Codex Programmaticus, Digital Equipment Corporation, Internal Publication, (1974).

(18) Spier M J, The Teleological Heap, in preparation.

(19) Spier M J, Programming Concepts, Invited Lecture, Spring DECUS Conference, Boston, May 29-31, 1974. (Working paper and Lecturing Notes; unpublished).

(20) Spier M J, Organick E I, The Multics Interprocess Communication Facility (revised), in Software Systems Principles: A Survey, Peter Freeman (Ed.), January 1975.

(21) Spier M J, A System Theoretic Look at the Complexity of Access Control Mechanisms, Proc. International Workshop on Protection in Operating Systems, Iria-Laboria, France, August 1974, pp. 225-241.

APPENDICE 2: BIBLIOGRAPHIE RELATIVE A CET OUVRAGE

La bibliographie suivante est donnée par ordre alphabétique de noms d'auteurs; par la nature de cet ouvrage, cette bibliographie est interdisciplinaire, s'étendant au delà des limites de l'informatique propre.

(1) Allport F H, Theories of Perception and the Concept of Structure, John Wiley, New York.

(2) Ashcroft E, Manna Z, The Translation of GOTO Programs to WHILE Programs, Technical Report STAN-CS-71-188, Computer Science Department, Stanford University, 1971.

(3) Baker F T, Chief Programmer Team Management of Production Programming, IBM Systems Journal vol. 11 #1, January 1972, pp. 56-73.

(4) Balzer R M, Dataless Programming, Proc. AFIPS 1967 Fall Joint Computer Conference, pp. 535-544.

(5) Balzer R M, Ports: A Method for Dynamic Inter-program Communication and Job Control, Technical Report R-605-ARPA, Rand Corporation, 1971.

(6) Beckman M, et al, Advanced Course in Software Engineering, Lecture Notes in Ecomonics and Mathematical Systems #81, Springer Verlag, New York, 1973.

(7) Beer S, The Surrogate World We Manage, Behavioral Science vol. 18 #3, 1973, pp. 198-209.

(8) Bertalanffy L von, General System Theory, George Braziller, New York, 1968.

(9) Bertalanffy L von, The History and Status of General System Theory, in Trends in General System Theory, Klir G J (Ed.), John Wiley, New York, 1972.

(10) Buxton J N, Randell B (Eds.), Software Engineering Techniques, Report on NATO Conference, Rome, October 1969.

(11) Cook S A, The Complexity of Theorem Proving Procedures, Proc. 3rd ACM Symposium on Theory of Computing, May 1971, pp. 151-157.

(12) Dahl O, Hierarchical Program Structures, in Structured Programming, Academic Press, New York, 1972.

(13) Dijkstra E W, A Constructive Approach to the Problem of Program Correctness, BIT vol. 8 #3, 1968, pp. 174-186.

(14) Dijkstra E W, GOTO Statement Considered Harmful, CACM vol. 11 #3, March 1968, pp. 147-148.

(15) Dijkstra E W, Notes on Structured Programming, in Structured Programming, Academic Press, New York, 1972.

(16) Floyd R W, Assigning Meanings to Programs, Proc. 19th AMS Symposium on Applied Mathematics, 1967, pp. 19-32.

(17) Floyd R W, Nondeterministic Algorithms, ACM Journal vol. 14 #4, 1967, pp. 636-644.

(18) Gileadi A, A Proposed Measure of Program Complexity, PhD Thesis Proposal, Computer and Information Sciences Dept., University of Massachusetts, Amherst, 1974.

(19) Good D I, Toward a Man-Machine System for Proving Program Correctness, PhD Thesis, University of Wisconsin, 1970.

(20) Gross J L, Brainerd W S, Fundamental Programming Concepts, Harper and Row, New York, 1972.

(21) Hamilton M, Zeldin S, Top-Down, Bottom-Up Structured Programming and Program Structuring, CSDL E-2728, December 1972.

(22) Henderson P, Snowdown R, An Experiment in Structured Programming, BIT vol. 12 #1, 1972, pp. 38-53.

(23) Hoare C A R, A Note on The FOR Statement, BIT vol. 12, 1972, pp. 334-341.

(24) Hoare C A R, Proof of a Program: FIND, CACM vol. 14 #1, 1971.

(25) Jones C B, Formal Development of Correct Algorithms: An Example Based on Earley's Recognizer, Proc. ACM Conf. on Proving Assertions About Programs, New Mexico State University, 1972, pp. 51-57.

(26) Johnson E, Can a Computer Adequately Administer a Problem-Solving Task to Individual Human Beings?, Technical Report 12, Dept. of Psychology, Yale University, 1966.

(27) King J C, A Program Verifier, PhD Thesis, Computer Science Dept., Carnegie Mellon University, Pittsburgh, 1969.

(28) Knuth D E, Floyd R W, Notes on Avoiding GOTO Statements, Technical Reports CS-148, Computer Science Dept., Stanford University, 1970.

(29) Knuth D E, Computer Science and Mathematics, American Scientist vol. 61, 1973, pp. 707-713.

(30) Laszlo E (Ed.), The Relevance of General System Theory, George Braziller, New York, 1972.

(31) Laszlo E, The Systems View of the World, George Braziller, New York, 1972.

(32) Laszlo E, A Systems Philosophy of Human Values, Behavioral Science, vol. 18 #4, 1973, pp. 250-259.

(33) Leavenworth B M, Programming with(out) the GOTO, Proc. ACM Annual Conf., Boston, 1972.

(34) Ledgard H F, Examples of Structured Programs, Computer and Information Science Dept., University of Massachusetts, Amherst, 1973.

(35) London R L, Proving Programs Correct: Some Techniques and Examples, BIT vol. 10 #2, 1970, pp. 168-182.

(36) McCarthy J, Towards a Mathematical Science of Computation, Proc. AFIP Congress, Amsterdam, 1962.

(37) Mills H D, Mathematical Foundations for Structured Programming, IBM Federal Systems Division, FSC 72-6012, 1972.

(38) Mills H D, The Complexity of Programs, IBM Federal Systems Division, 1972.

(39) Mills H D, Top-Down Programming in Large Systems, in Debugging Techniques in Large Systems, Rustin R (Ed.), 1st Courant Computer Science Symposium, New York University, 1971, pp. 41-55.

(40) Mills H D, How to Write Correct Programs and Know It, IBM Federal Systems Division, January 1973.

(41) Minsky M, Matter, Mind and Models, Proc IFIP Congress 1965, pp. 45-49.

(42) Naur P, The Place of Programming in a World of Problems, Pric. IFIP Congress 1965, pp. 195-199.

(43) Naur P, Programming by Action Clusters, BIT vol. 9, 1969, pp. 250-258.

(44) Naur P, Randell B (Eds.), Software Engineering, Report on NATO Conference, Garmish, 1969.

(45) Nilsson N J, Problem-Solving Methods in Artificial Intelligence, McGraw Hill, 1971.

(46) Parnas D L, A Technique for Software Module Specification with Examples, CACM vol. 15 #5, 1972, pp. 330-336.

(47) Parnas D L, On the Criteria to be Used in Decomposing Systems into Modules, CACM vol. 15 #12, 1972, pp. 1053-1058.

(48) Simon H A, The Architecture of Complexity, Proc. of the American Philosophical Society, 106, December 1962, pp. 467-482.

(49) Spinelli D N, OCCAM: A Computer Model for a Content Addressable Memory in the Central Nervous System, in Biology of Memory, Academic Press, New York, 1970

(50) Sussman G J, A Computational Model of Skill Acquisition, PhD Thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, August 1973.

(51) van Emden M H, An Analysis of Complexity, PhD Thesis, Mathematisch Centrum, Amsterdam, 1971.

(52) van Emden M H, The Hierarchical Decomposition of Complexity, in Machine Intelligence, Dale and Michie (Eds.), Edinburgh University Press, 1970, pp. 361-380.

(53) Wegner P, Three Computer Cultures: Computer Technology, Computer Mathematics and Computer Science, Technical Report 70-14, Center for Computer and Information Sciences, Brown University, 1970.

(54) Wegner P, Structured Programming, Program Synthesis and Semantic Definition, Technical Report 72-63, Center for Computer and Information Sciences, Brown University, 1972.

(55) Wirth N, Program Development by Stepwise Refinement, CACM vol. 14 #4, 1971, pp. 221-227.

(56) Wulf W A, A Case Against the GOTO, Proc. ACM Annual Conf., August 1972, pp. 791-796.

(57) Wulf W A, Shaw M, Global Variables Considered Harmful, ACM SIGPLAN Notices, vol. 8 #2, 1973.

(58) Zurcher F, Randell B, Iterative Multilevel Modelling: A Methodology for Computer System Design, Proc. IFIP Congress, 1968, pp. 138-142.